A C++ based MPI-enabled Tasking Framework to Efficiently Parallelize Fast Multipole Methods for Molecular Dynamics

Dissertation
zur Erlangung des akademischen Grades
Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Math. David Haensel
geboren am 31. März 1988 in Eberswalde-Finow

Gutachter:
Prof. Dr. Wolfgang E. Nagel, Technische Universität Dresden
Prof. Dr. Hans-Joachim Bungartz, Technische Universität München

Tag der Verteidigung:
02. Juli 2018

Dresden im August 2018

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API Application program interface

CAS Compare-and-swap

CRTP Curiously recurring template pattern

DAG Directed acyclic graph

FFT Fast Fourier Transform

FIFO First in – first out

FMM Fast Multipole Method

HPC High performance computing

JSC Jülich Supercomputing Centre

L2L Local to local

LIFO Last in – first out

M2L Multipole to local

M2M Multipole to multipole

MD Molecular dynamics

MPI Message passing interface

NUMA Non-uniform memory access

PGAS Partitioned global address space

PME Particle Mesh Ewald

RDMA Remote direct memory access

SFINAE Substitution failure is not an error

SIMD Single instruction, multiple data

SIMT Single instruction, multiple threads

SMT Simultaneous multithreading

TBB Thread building blocks

TMP Template-meta-programming

ULT User-level-threads

UMA Uniform memory access

ABSTRACT

Today's supercomputers gain their performance through a rapidly increasing number of cores per node. To tackle issues arising from those developments new parallelization approaches guided by modern software engineering are inevitable. The concept of task-based parallelization is a promising candidate to overcome many of those challenges. However, for latency-critical applications, like molecular dynamics, available tasking frameworks introduce considerable overheads. In this work a lightweight task engine for latency-critical applications is proposed. The main contributions of this thesis are a static data-flow dispatcher, a type-driven priority scheduler and an extension for communication-enabled tasks. The dispatcher allows a user-configurable mapping of algorithmic dependencies in the task-engine at compile-time. Resolving these dependencies at compile-time reduces the run-time overhead. The scheduler enables the prioritized execution of a critical path of an algorithm. Additionally, the priorities are deduced from the task type at compile-time as well. Furthermore, the aforementioned task engine supports inter-node communication via message passing. The provided communication interface drastically simplifies the user interface of inter-node communication without introducing additional performance penalties.

This is only possible by distinguishing two developer roles – the library developer and the algorithm developer. All proposed components follow a strict guideline to increase the maintainability for library developers and the usability for algorithm developers. To reach this goal a high level of abstraction and encapsulation is required in the software stack. As proof of concept the communication-enabled task engine is utilized to parallelize the FMM for molecular dynamics.

CHAPTER 1

INTRODUCTION

Computational science has become "The 'Third Pillar' of 21st Century Science" [100, pp. 12] positioned between theory and experiment blurring the line between both of them. Several scientific breakthroughs like the decoding of the human genome [45] or the space missions to Mars [116] would have been impossible without computational science and computer simulations. Fields like drug research utilize computer simulations to discover new use cases and allow to predict unknown side effects. For almost all fields of science, computer simulations have become virtually indispensable. The resources required for those computer simulations form the rising field of high performance computing (HPC). The rapid increase in affordable computational resources and powerful hardware enables simulations that have been unthinkable decades ago. Nowadays, large scale simulations required for the weather forecast [24, 119] or simulations of quantum computers [120] are only possible because supercomputers offer a vast amount of computational power. This development took on momentum and with exascale in sight [79] further scientific insights will be discovered by an increasing field of applications.

However, utilizing exascale hardware to the full extend will be challenging. In the past, supercomputers gained most of their performance by adding more nodes to the system. Additionally, an almost free performance increase came from the raised clock-speed of the processors and shrinking die sizes due to Moore's law [84]. A few years into the new millennium the clock-speed stagnated and further performance improvements were only possible through the advent of multicore processors. This also changed the subsequent development of supercomputers. Instead of increasing the performance by increasing the numbers of nodes, the nodes themselves became more powerful (see Figure 1.1). For the ongoing development towards exascale this means, the number of cores per node will increase rapidly, whereas the number of nodes will increase only moderately [11]. This inevitable leads to a more complex node. Additionally, new and wider vector instructions or the introduction of non-uniform memory access (NUMA) contribute to the complexity as well. Besides classical CPUs also new architectures like GPUs or other accelerators emerged, requiring completely differ-

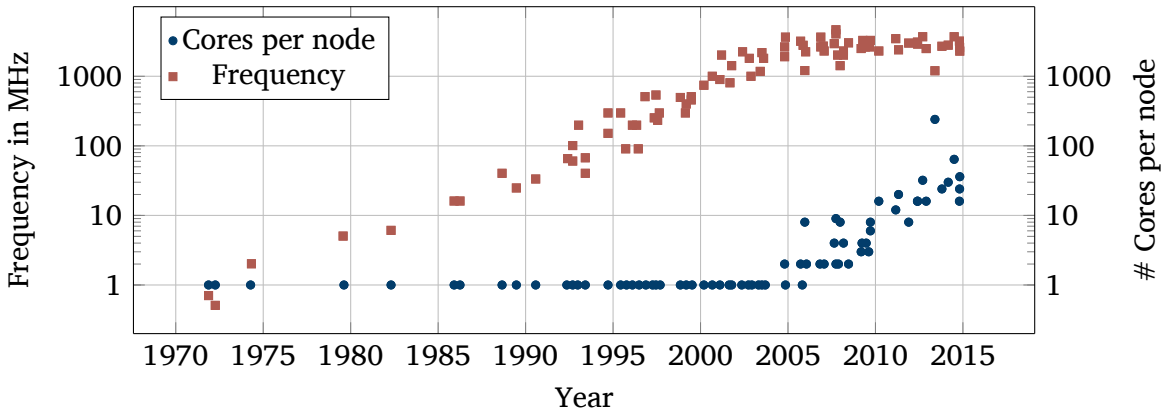


Figure 1.1: A comparison of the development of the CPU frequency and the number of cores per node in the TOP500 list [111].

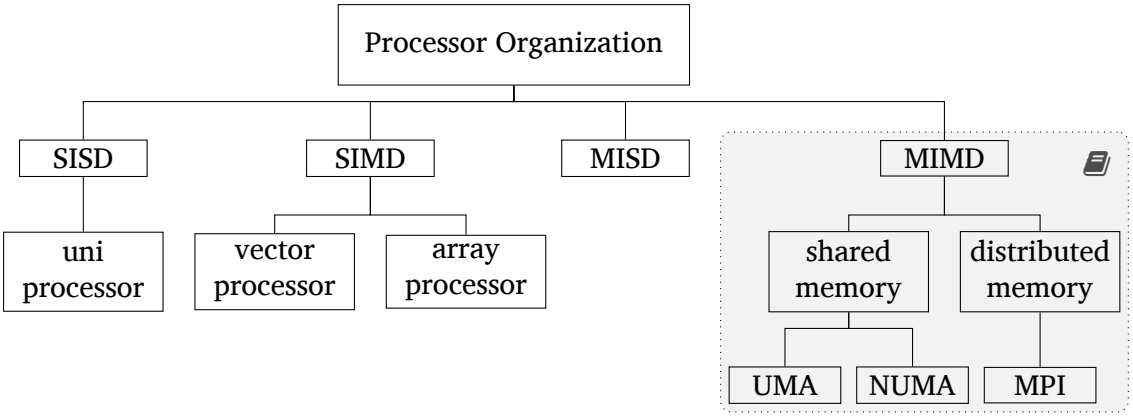


Figure 1.2: Flynn's Taxonomy and Johnson extension for MIMD architectures.

ent programming models. Referring to Flynn's extended taxonomy [40, 67] this prolonged trend holds parallelization potentials on many levels of the available hardware (see Figure 1.2).

The overall development of HPC led to many diverse systems already today, offering quadrillion (10^{15}) of floating-point operations per second. For systems passing the exascale with quintillion (10^{18}) of floating-point operations the hardware will become even more complicated in comparison [4]. These developments are ubiquitous in HPC and will also enforce changes on the software development. This work will contribute to the handling of rapidly developing hardware and focuses on the software engineering challenges.

How did the software development for HPC evolve? For the parallelization, the first supercomputer applications used only inter-node communication techniques to distribute the data between nodes in an otherwise sequential program. With the advent of multicore processors shared memory parallelization was required for the first time. Exploiting loop-level parallelism by sprinkling pragmas throughout

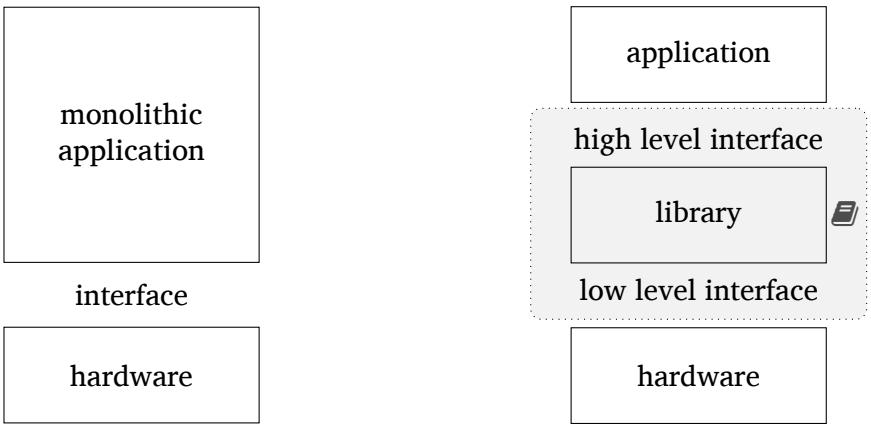


Figure 1.3: The separation of concerns splits the monolithic application into a high level and a low level layer.

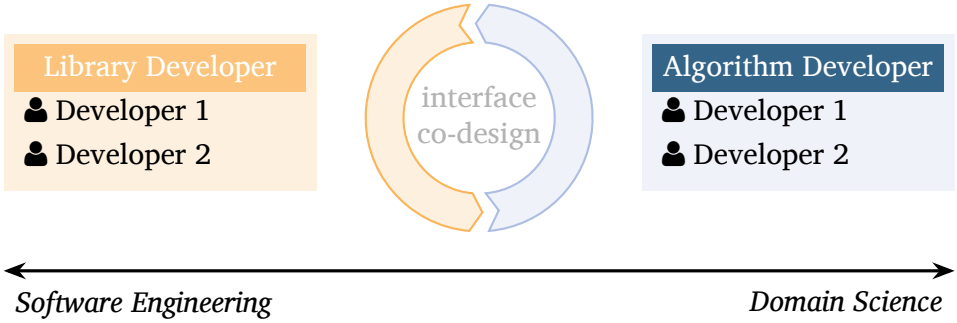


Figure 1.4: The distinguishing of two developer roles in HPC. The library developer is responsible for the low level library and hardware abstraction. The library developer provides an easy-to-use interface to the algorithm developer. The algorithm developer is responsible for the algorithm and the implementation of the algorithm using the high level interfaces provided by the library developer.

the sequential code at each loop construct solved the problem initially. However, this led to programs with millions of lines of code, being hardly maintainable and unable to adapt to even more sophisticated hardware features like vectorization.

To remain sustainable modern software requires abstractions of hardware in a more generic manner. Additionally, the separation of concerns between algorithm-specific parts of the program and the parallelization and hardware specific parts should be enforced (see Figure 1.3). In the past, for most programs a single domain scientist was responsible for everything: the algorithm, implementation, parallelization, optimization and porting. With the rapid development of new hardware and the thereby increased complexity, this is not manageable by a single software developer anymore. Therefore, the proposed separation of concerns in the software must also be considered for the software developers themselves.

This can be achieved by distinguishing two different software developer roles

in HPC (see Figure 1.4):

The Library Developer is responsible for the low level libraries encapsulating hardware features and parallelization. The library developer is also responsible for providing a high level interface which can be used by the algorithm developer that ideally hides most of the non-performance-critical features.

The Algorithm Developer is responsible for the high level algorithmic details and the implementation of the algorithm using the high level interfaces provided by the library developer.

Algorithmic development should be independent of specific hardware. This differentiation allows the individual developer to focus on the challenges specifically arising in their field of expertise. The challenges in providing parallelization tools and low level implementations are shifted towards computer science. Simultaneously it relieves the library developer from the responsibility of fully understanding algorithmic details and vice versa.

Even though this work focuses on the library developer side of the classification a use case for the proposed concepts is required. For this purpose the Fast Multipole Method (FMM) for solving long range interactions in molecular dynamics (MD) was chosen. This method exhibits parallelization challenges already today that other application might not experience until the exascale era. In MD the particle movement of a fixed particle system is simulated usually encompassing thousands to millions of particles. The simulation is done in discrete time steps each evaluating the long range potentials and forces using the FMM. For a reasonable simulation time millions of time steps are necessary. To avoid an excessive overall runtime, the single step runtime of the FMM must therefore be in the range of only a few milliseconds. To reach that goal massive parallelism needs to be exploited and therefore each compute unit (core) will only hold a few particles. But for a few particles per core only very little computations per core are required and data transfer and communication overheads become apparent. These kind of problems will be called latency-critical problems in this work.

To perform the parallelization of latency-critical problems, the library developer needs to provide a programming model suited for the task at hand. The programming model of this work is restricted to the shared memory parallelization and distributed parallelization in Flynn's taxonomy (see Figure 1.2). For the parallelization a coupled solution for both problems will be presented. This stands in contrast to all-inclusive solutions for inter-node and intra-node parallelization like HPX [70], Charm++ [71], PaRSEC [19] and DAGuE [18]. These libraries work well for coarse-grained tasks encompassing a sufficient number of computations. However, for latency-critical problems fine-grained tasks must be used and those approaches do not provide sufficient control to adapt the parallelization layer to the requirements of the algorithm.

For the parallelization of latency-critical application it is necessary to control the program flow on all levels. Therefore, the parallelization concepts and tools

proposed within this work deal with intra-node and inter-node parallelization separately. In the chapter on intra-node parallelization current state of the art parallelization approaches will be discussed and a task engine specialized for latency-critical problems will be proposed. This task engine includes two unique components not available in other frameworks, namely the static data-flow dispatcher and the type-driven priority scheduler. The static data-flow dispatcher is compile-time configurable with algorithmic dependencies. The type-driven priority scheduler, schedules tasks using different priorities which are automatically deduced from the type of the task.

For the inter-node parallelization existing approaches and libraries will be discussed in the chapter inter-node parallelization. There, the task engine will be extended with a communication layer. This communication layer simplifies the communication interface for message passing and provides an easy-to-use high level interface for the algorithm developer. Additionally, initial concepts for task-based communication using fine-grained tasks will be shown.

Nomenclature

For the remainder of this work the following nomenclature is used: A supercomputer consists of several compute nodes, which are connected with a fast network. A compute node has one or more sockets. A processor refers to a single socket CPU. This processor consists of one or more cores. On a core one or more hardware-threads are executed, depending on simultaneous multithreading (SMT).

1.1 Design Guidelines

The implementation follows several well-established software design guidelines and principles. The following four design goals are the most important ones:

- Correctness,
- Maintainability,
- Sustainability and
- Performance portability.

Maintainability

For the long-term use of a library it is very important to provide a maintainable and readable code base. Without this requirement, it is hard to extend the library at a later point in time or adapt to new circumstances. During the implementation phase this guideline has been enforced with test-driven development [75].

An additional strategy is to stick to well established coding principles. One famous principle is the object oriented design principle SOLID [77]. SOLID consists of the following five sub-principles:

The single-responsibility principle (S) states, that every class or module should only have a single responsibility. This results in code with a high level of encapsulation.

The open/closed principle (O) states, that the software should be "open for extension, but closed for modification" [80]. This means, that changes to a component or a class should only be possible by extension like inheritance, but not by changing the existing implementation itself.

The Liskov substitution principle (L) states, that properties valid for one type should still be valid when exchanging this type with its sub-type [76].

The interface segregation principle (I) states, small interfaces are better than big ones and thereby often too general interfaces.

The dependency-inversion principle (D) suggest to better depend on abstractions instead of concrete classes.

These principles already require code with a high level of abstraction and encapsulation. Additionally, the "Don't repeat yourself" (DRY) principle [59] was followed in the implementation.

Performance Portability

For applications in HPC it is expected to show good performance on the latest hardware platform. Due to the frequent changes and availability of diverse supercomputers, nowadays it is required to exhibit convincing performance on many different platforms. This requirement is herein called performance portability.

As described before the computing hardware changes in many directions. To handle quantitative changes in hardware effectively a high level of abstraction is required. For legacy software written in Fortran or C this means a lot of work porting and optimizing code to new supercomputers. Adapting to all changed hardware features and optimizing historically grown software by hand is time consuming and potentially prone to errors.

This problem can be partially solved by abstracting away the target hardware in an generic manner. Using template-meta-programming (TMP) features of C++ this can be achieved without constantly porting software by hand. Such an approach will pose a challenge to library developers, but ultimately will help domain scientists to harvest the performance of the target hardware without the need to understand hardware details. The herein proposed abstractions come with zero runtime overhead and are to this extend only possible using C++ and TMP. This makes C++ a first choice for solving the challenges for the programming of current and future supercomputers.

1.2 HPC and C++

Parallelization of supercomputers cannot be discussed without putting it in context to the used programming language. The most common programming languages supported on supercomputers are Fortran, C and C++, whereas Fortran an C are the most prevailing [121, 82]. Only a few application are using C++ and when it comes to modern C++ with template-meta-programming, the numbers are even lower.

This fact is surprising since C++ has a lot to offer – especially for HPC. C++ is a mature industrial standard [66] with an established standardization process and a vivid community. The most important extension of the language for parallel computing was added in the 2011 standard. This extension concerned the memory model and the multithreaded execution capabilities directly inscribed in the language [16, 118]. Before the availability of this extension, non of the languages used in HPC were inherently parallel. It was required to use external libraries like pthreads [89] but this raised thread safety concerns [15].

The upcoming HPC technologies are not uniform anymore and the HPC ecosystem will be more diverse in future. As mentioned before, this requires a high level of abstraction in order to reach performance portability more easily and C++ has a solution for that as well. Zero-overhead abstraction [49] is one of the main design principles of the C++ language itself. As the founder of C++ B. Stroustrup said [106]:

> In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

Zero-overhead abstraction refers specifically to inheritance and template-metaprogramming (TMP). This work is heavily based on modern C++ features and especially on TMP abstraction capabilities. In the following the foundational concepts and special language features required for the intra-node and inter-node parallelization are laid out.

Compile-Time Abstraction

Zero-overhead abstractions have been possible by using pre-processor macros in C, Fortran and C++. Besides pre-processor macros, C++ additionally offers templates. Both, C++ templates and pre-processor macros are evaluated at compile time. But in contrast to pre-processor macros, templates are part of the type system of C++. This means, that potential bugs that could arise at runtime or stay undetected, will be identified at compile-time. This increases the robustness of the code [107] tremendously.

Thus, besides the increased performance, maintainability and increased robustness are the main reasons for using C++ templates. Since it is possible to have many fine-grained layers of software designed in such a way, true separation of concerns becomes possible.

Generic Programming

Generic programming describes the term of program code being independent of the actual type. This is well known from generic data structures like the vector in the C++ standard template library. The implementation of those data structures is available independently of the concrete type. Instead of concrete types, placeholders are used. These placeholders are called template parameters. These data structures can be used by applying the desired type. The compiler will

generate the code by replacing the placeholder with the desired type. Since the template parameters are replaced by concrete types at compile-time this comes without runtime overhead. This makes generic programming to an preeminent zero cost abstraction feature.

C++ offers class and function templates. These functions or classes may have template parameters. Template parameters can either be types or non-type template parameter like integers. Since C++11 it is also possible to use template packs, which is an arbitrary number of template parameters defined in a single pack.

Template Specialization

The original use case of templates in C++ was generic programming. Template-meta-programming (TMP) itself and the Turing completeness [115] of TMP were discovered by accident. However, the mature feature set and abstraction capabilities of TMP makes it indispensable for this work. Not utilizing TMP would require to implement a lot of compiler generated code by hand in order to reach a comparable performance and portability. This would sweep away the configurability of any library developed today.

Template specialization is the first step towards template-meta-programming (TMP). Function templates as well as class templates can be fully specialized. This means, the specialized implementation can be provided for a certain combination of template parameters. This is especially helpful for implementing generalized compute kernels not depending on a specific floating point type.

Additionally, class templates have the capability of partial template specialization. In contrast to the full template specialization, only parts of the template parameter list needs to be specialized. The remainder of the template parameters are still generic.

Compile-Time Branching

With a classical *if-then-else*-condition a branch of the control-flow can be determined. If it is possible to evaluate the condition at compile-time, it would be ideal to choose the branch at compile-time. With C++ and TMP this is possible. Therefore, compile-time branching is an important feature for the implementation of reusable code.

One concept facilitating compile-time branching is “substitution failure is not an error” (SFINAE). The concept of SFINAE can be described as follows: An invalid substitution of a template parameter during function overload resolution does not result in an compile-time error, but rather discards this function definition. This means, the same function could be implemented multiple times with different template parameter definitions. For calling the function template, the template parameters need to be applied either explicitly or by template argument deduction. During the overload resolution the applied parameters are substituted in all function definitions. The substitution may lead to an ill-formed function definitions, causing a substitution failure. However, this failure is not an error and the function will be discarded from the overload resolution.

LISTING 1.1: EXAMPLE OF COMPILE-TIME BRANCHING

```
1 // function implementation for true condition
2 template <typename T>
3 typename std::enable_if<condition, void>::type foo() {...}
4
5 // function implementation for false condition
6 template <typename T>
7 typename std::enable_if<!condition, void>::type foo() {...}
```

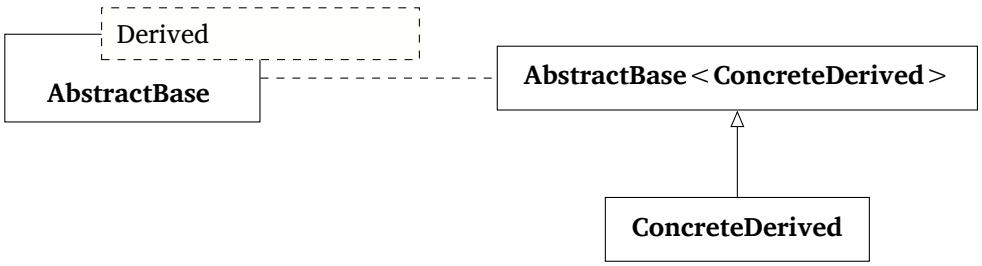


Figure 1.5: The UML class diagram of static inheritance using CRTP.

It is important, that only one function definition remains, otherwise the function definition would be ambiguous.

For the concrete implementations the type trait `std::enable_if` can be used. An example using SFINAE and `enable_if` is shown in Listing 1.1. In case the condition used evaluates to `true` the first function implementation is used (see line 3), otherwise the second implementation will be executed (see line 7).

STATIC INHERITANCE

Static inheritance could be used to share behavior between classes without introducing virtual inheritance overhead. The advantage is the reduction of code duplication and hence an increased robustness and maintainability. One TMP technique enabling static inheritance is called curiously recurring template pattern (CRTP).

For CRTP a derived class inherits from a common templated base class (see Figure 1.5). Listing 1.2 shows an example of CRTP. The base class (`AbstractBase`)

LISTING 1.2: EXAMPLE OF CURIOUSLY RECURRING TEMPLATE PATTERN

```
1 template <typename Derived>
2 struct AbstractBase {
3   Derived & derived() {
4     return *static_cast<Derived *>(this);
5   }
6   void virtual_method() {
7     derived().virtual_method();
8   }
9 };
10
11 struct ConcreteDerived : AbstractBase<ConcreteDerived> {
12   void virtual_method() {}
13 };
```

uses a template parameter (`Derived`) which reflects the derived class. This means, strictly speaking derived classes do not have a common base class due to their different template parameters. In the base class it is possible to statically cast the object towards an instance of the derived class shown in the method `derived()`. With the help of such a cast, shared behavior can be implemented in the base class. By implementing the same method in the derived class, the method of the base class can be hidden. Additionally, methods behaving like virtual inheritance can be implemented. In the example given in the listing, the method `virtual_method` calls the implementation of the derived class. This is done by calling the method on the derived class using the cast method `derived()`.

1.3 Molecular Dynamics

Molecular dynamics (MD) simulations characterize the evolution of a particle system over time. Therefore, the movement of particles due to their interactions is simulated. MD has a long history in computational science starting from the late sixties [3]. There are several MD implementations like Gromacs [56], NAMD [88] or Amber [22] available today. Due to the high demand in computational resources, also special purpose hardware was developed to compute MD simulations like MD Grape [46] and D.E. Shaw's Anton [103].

The simulation itself is performed in discrete time steps. These time steps are computed in the so called MD-loop which is shown in a simplified diagram in Figure 1.6. In general an MD loop can be described as follows. The long-range and short-range interactions (potentials and forces) are computed. The computed forces are used to determine the new particle positions in the next time step. Then, the process starts over again using the new particle coordinates. It should be mentioned, that the loop introduces an implicit synchronization. The integration can only take place after all interactions are computed.

The simulated particle systems usually consist of hundred thousands to several million of charged particles. These particles might resemble proteins, DNA or even bio-molecules like whole viruses [124]. Since the common time step required for those MD simulations is in the range of femtoseconds and the desired total simulation time is required to be in the micro- or even milliseconds, the total number of time steps therefore will be in the millions to billions. To maintain a

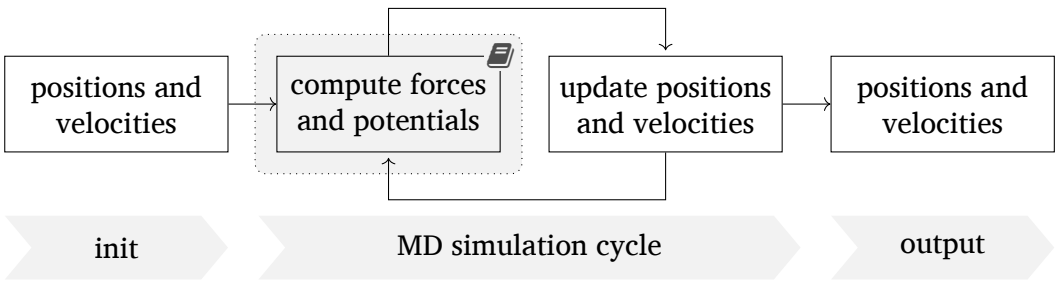


Figure 1.6: The simplified version of the internal MD Loop.

reasonable total runtime the execution time required for a single time step needs to be in the lower millisecond range. Since the number of particles can not be increased arbitrarily, weak-scaling is not an option and the development of MD has to focus on strong-scaling. This already imposes challenges in the parallelization on current petascale hardware.

This work contributes to a software module for the MD simulation toolbox Gromacs and is embedded in the code development of the SPPEXA [21] project GROMEX [53]. Gromacs is an open source software developed in Europe. It has a large user-base of several thousand scientists in academia and research. Additionally, the project has been included in the folding@home community extending the compute power to thousands of clients [101]. Such a large and diverse user-base requires the code to run on a multitude of hardware ranging from a student's laptop to the latest supercomputer.

One might ask, what is the most time consuming part of such MD simulations? Since the integration step is embarrassingly parallel and has no inter-particle dependencies it only takes a tiny portion of the overall runtime. The overwhelming part of the runtime is consumed by the computation of the long-range interactions. Since all mutual interactions have to be computed, the naïvely exhibited computational complexity is $\mathcal{O}(N^2)$ with respect to the number of particles N. With the help of Particle Mesh Ewald (PME) this complexity can be reduced to $\mathcal{O}(N \log N)$. Nevertheless, the significant part of runtime still stems from the long-range interactions. Additionally, PME itself has another drawback. Since it employs a Fast Fourier Transform (FFT) it has an inherent parallel scaling bottleneck due to global communication requirements of such FFTs. The contributed software module exchanges the PME algorithm with the FMM for the computation of long-range interactions.

Chapter 2

The FMM in a Nutshell

The Fast Multipole Method (FMM) is a numerical method for solving the N-body problem. The N-body problem is about the computation of all pairwise interactions between N particles. Those particles might be charges interacting due to electrostatic forces or masses interacting due to gravitational forces. This work focuses especially on the computation of the Coulomb potential arising in molecular dynamics.

Definition 2.1 *The **Coulomb potential** at target position $\boldsymbol{x}_i$ for N source particles at positions $\boldsymbol{x}_j \in \mathbb{R}^3$; $i,j \in \{1,\dots,N\}$ and the corresponding charges q_j is defined as:*

$$\Phi\left(\boldsymbol{x}_i\right) = \sum_{j=1}^{N} \frac{q_j}{\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|_2} \quad (i \neq j) \ . \tag{2.1}$$

The corresponding force at target particle at position $\boldsymbol{x}_i$ with charge q_i is defined as:

$$\boldsymbol{F}\left(\boldsymbol{x}_i\right) = q_i \sum_{j=1}^{N} \frac{q_j}{\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|_2^3} \left(\boldsymbol{x}_i - \boldsymbol{x}_j\right) \quad (i \neq j) \ . \tag{2.2}$$

As seen in Equation 2.1, the computation of a single potential $\Phi(\boldsymbol{x}_i)$ has linear complexity. Since potentials should be evaluated for N particles, the N-body problem exhibits a computational complexity of $\mathcal{O}(N^2)$. This complexity is unfavorable for large simulations since the computation time increases rapidly and thereby limits the size of the simulation. There are several fast summation schemes like particle mesh Ewald [31] and Barnes-Hut [12] reducing the complexity to $\mathcal{O}(N \log N)$ and multigrid methods [112] even exhibiting only linear complexity.

The Fast Multipole Method (FMM) also exhibits linear complexity. It was first described in 1987 by Greengard and Rokhlin [50]. This work discusses a FMM specialized for MD focusing only on the Coulomb potential. Nevertheless, many other FMM implementations exist for the Coulomb or gravitational potential like ExaFMM [123] or ScalFMM [2]. However, those implementations lack optimization and specialization for the constraints introduced by the MD use case like

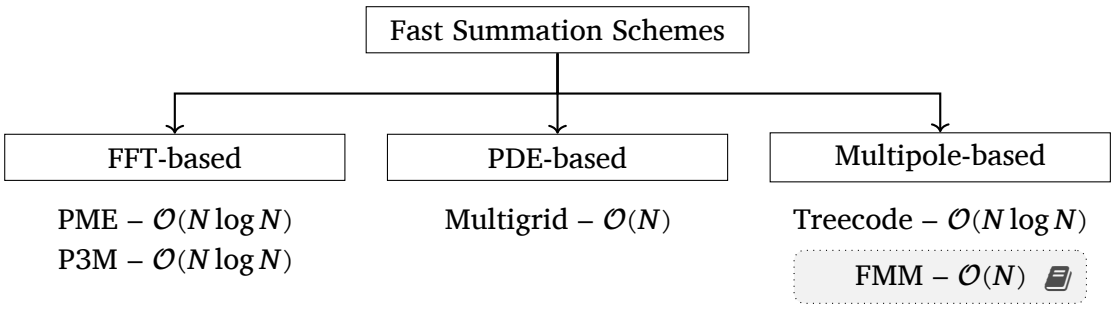


Figure 2.1: Computational complexity of different fast summation schemes with reduced complexity.

homogeneous distributed particles and execution time close to a millisecond. Additionally, other FMM implementations have been proposed for arbitrary kernels in a kernel independent way like KIFMM [122].

The main idea of the FMM is the grouping of far distant particles into expansions. These expansions are used for the computation of pseudo interactions between the groups instead of the computation of every single interaction between particles of these groups. The following notation and derivation is adapted from the FMM proposed in [69].

2.1 Particle Grouping

To get a better understanding on how the FMM works, the particle distribution shown in Figure 2.2a is assumed. The particles are positioned in a way, that no overlapping of the spheres containing the particles can occur. The first group contains M particles and the second group contains N particles. If the interactions between particles inside a sphere are ignored, this results in $M \cdot N$ interactions.

As long as the distance between the two groups is big enough, a change of the particle position in one group does only slightly influence the computed potentials in the other group. This means, instead of computing all $M \cdot N$ interactions, only one pseudo interaction between both groups can be computed (see Figure 2.2b). This pseudo interaction was not introduced yet, but a first approximation could be to sum up all charges of particles inside the group. This sum can be assigned

(a) $M \cdot N$ interactions (b) one interaction

Figure 2.2: Grouping and interaction of far distant particles for classical direct interaction via particles and interaction via pseudo-particles (multipoles).

to a pseudo particle at the center of the group. This interaction would be called a monopole interaction. Taking higher order expansions like dipoles, quadrupoles, octopoles or hexapoles into account increases the accuracy. Obviously, this approach introduces errors since the expansion must be finite. However, there exists an expansion order that corresponds to the precise (direct) computation of the system. Since other errors are introduced by the MD simulation it is sufficient to use a lower precision requirement for the FMM as well. This will speed up the computation even more.

2.2 Multipole and Local Expansions

For the computation of the forces and the potentials, the FMM utilizes two different expansions. The first is the multipole expansion and the second is the local expansion. The derivation can be performed in Cartesian coordinates, however, using spherical coordinates results in a more compact representation. In order to derive the spherical expansions additional polynomials are required:

Definition 2.2 *The **Legendre polynomials** with $l \in \mathbb{N}_0$ are defined as:*

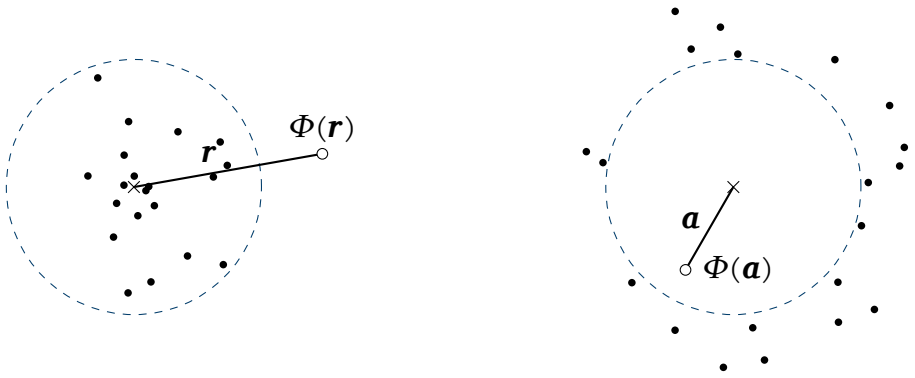
$$P_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} \left[\left(x^2 - 1\right)^l \right] .$$

Definition 2.3 *The **associated Legendre polynomials** with $m \in \mathbb{Z}$ are defined as:*

$$P_{lm}(x) = (-1)^m \left(1 - x^2\right)^{m/2} \frac{d^m}{dx^m} P_l(x) \quad (|m| \leq l) .$$

Definition 2.4 *A complex coefficient $\omega_{lm} \in \mathbb{C}$ of the **multipole expansion** ω of k particles with position $\boldsymbol{a}_j = (a_j, \alpha_j, \beta_j), j \in \{1, \ldots, k\}$ inside a sphere with radius $\hat{a}$ and center $\boldsymbol{a} = (a, \alpha, \beta)$ with $a_j < \hat{a}$ and corresponding charges q_j is defined as:*

$$\omega_{lm}(\boldsymbol{a}) = \sum_{j=1}^{k} \frac{q_j a_j^l}{(l+m)!} P_{lm}\left(\cos \alpha_j\right) e^{-im\beta_j} .$$



(a) Multipole Expansion (b) Local Expansion

Figure 2.3: Multipole expansion and local expansion with corresponding potential Φ.

Definition 2.5 *A complex coefficient $O_{lm}(\boldsymbol{a}) \in \mathbb{C}$ of the **chargeless multipole expansion** of a point $\boldsymbol{a} = (a, \alpha, \beta)$ is defined as:*

$$O_{lm}(\boldsymbol{a}) = \frac{a^l}{(l+m)!} P_{lm}(\cos\alpha)\, e^{-im\beta}.$$

Assuming a point $\boldsymbol{r} = (r, \theta, \phi)$ with $r > \hat{a}$ outside the sphere of the multipole (see Figure 2.3a), the potential at this point introduced by the particles represented by the multipole expansion $\omega(\boldsymbol{a})$ is given by:

$$\Phi(\boldsymbol{r}) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \omega_{lm}(\boldsymbol{a}) \frac{(l-m)!}{r^{l+1}} P_{lm}(\cos\theta)\, e^{im\phi}. \tag{2.3}$$

The multipole expansion is valid for the computation of the potential for all points outside the sphere. The convergence with respect to the expansion length of the potential increases for increasing distance between the sphere and the evaluation point ($r \gg a$).

Definition 2.6 *A complex coefficient $\mu_{lm}(\boldsymbol{r}) \in \mathbb{C}$ of the **local expansion** μ expanded at $\boldsymbol{r} = (r, \theta, \phi)$ with radius $\hat{r}$ and k particles with position $\boldsymbol{r}_j = (r_j, \theta_j, \phi_j), j \in \{1, \dots, k\}$ with $r_j > \hat{r}$ and corresponding charges q_j is defined as:*

$$\mu_{lm}(\boldsymbol{r}) = \sum_{j=1}^{k} q_j \frac{(l-m)!}{r_j^{l+1}} P_{lm}\left(\cos\theta_j\right) e^{im\phi_j}. \tag{2.4}$$

Definition 2.7 *A complex coefficient $_{lm}(\boldsymbol{r}) \in \mathbb{C}$ of the **chargeless local expansion** of a point $\boldsymbol{r} = (r, \theta, \phi)$ is defined as:*

$$M_{lm}(\boldsymbol{r}) = \frac{(l-m)!}{r^{l+1}} P_{lm}(\cos\theta)\, e^{im\phi}.$$

Let's assume another point $\boldsymbol{a} = (a, \alpha, \beta)$ with $a < \hat{a}$ inside the sphere (see Figure 2.3b). The potential at this point introduced by the particles outside the sphere represented by the local expansion μ is given by:

$$\Phi(P) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \mu_{lm}(\boldsymbol{r}) \frac{a^l}{(l+m)!} P_{lm}(\cos\alpha)\, e^{-im\beta}. \tag{2.5}$$

In contrast to the multipole expansion, this formula is only valid for evaluation points inside the sphere and the convergence becomes better for points close to the center of the expansion ($a \ll r$). Equation 2.5 and Equation 2.3 show, that the outer sum extends to infinity. For the computation the series must be truncated at a finite $p \in \mathbb{N}_0$, called the *multipole order*. This truncation introduces an error which can be controlled. There exist error estimation schemes, but as a general rule of thumb a higher order expansion leads to a smaller error in the computation.

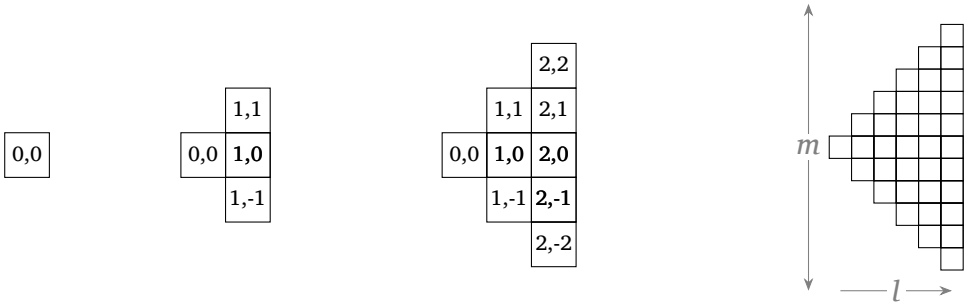


Figure 2.4: Triangular structure of a multipole or local expansion for a monopole, a dipole, a quadrupole and a multipole with multipole order $p = 5$.

Additionally, it can be observed that, the series uses a double index l, m. For the truncated formula this leads to a triangular-like structure of the used multipole expansion coefficients (see Figure 2.4). Furthermore, multipole expansions expanded at the same center can be summed up. In this case, the radius of the new multipole expansion is the larger of both radii. The same summation rule is valid for local expansions at the same center. However, for local expansions the smaller radii is the radius of the new expansion.

2.3 FMM Operators

Until now, only multipole expansions or local expansions at a certain center can be computed. For the FMM algorithm it is of interest, to compute expansions directly from other expansions without using particles in every step. Operators facilitating such possibilities are shown in the following sections.

2.3.1 Multipole to Multipole

Definition 2.8 *The coefficients A_{jk}^{lm} of the **multipole to multipole** operator are defined as:*

$$A_{jk}^{lm}(\boldsymbol{b}) = O_{l-j,m-k}(\boldsymbol{b}) \ .$$

The multipole to multipole (M2M) operator can be used to shift a multipole from one center to another center without expanding particles again. This means, a

Figure 2.5: The multipole to multipole operator allows to shift multipoles around an old center to a new center.

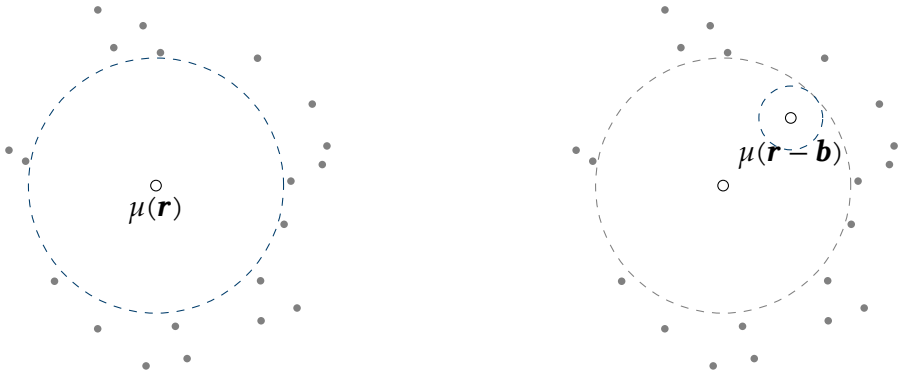


Figure 2.6: The local to local operator allows to shift a local expansion around an old center to a new center inside the sphere.

multipole $\omega(\boldsymbol{a})$ expanded at the center $\boldsymbol{a}$ can be shifted to a new center $\boldsymbol{a}+\boldsymbol{b}$ (see Figure 2.5). The coefficients of the new multipole $\omega(\boldsymbol{a}+\boldsymbol{b})$ can be computed as follows:

$$\omega_{lm}(\boldsymbol{a}+\boldsymbol{b}) = \sum_{j=0}^{l}\sum_{k=-l}^{l} \omega_{jk}(\boldsymbol{a})\, A_{jk}^{lm}(\boldsymbol{b}) \ . \tag{2.6}$$

This operation does not introduce additional errors itself. The result of applying the operator is the same as if the particles were expanded directly at the new center. As seen in Equation 2.6 the computation of a single coefficient of the multipole expansion $\omega(\boldsymbol{a}+\boldsymbol{b})$ exhibits a complexity of $\mathcal{O}(p^2)$. Since the multipole expansion has $(p+1)(p+2)/2$ complex coefficients, the complexity for the computation of all coefficient is $\mathcal{O}(p^4)$.

2.3.2 Local to Local

Definition 2.9 *The **local to local** operator C_{jk}^{lm} is defined as:*

$$C_{jk}^{lm}(\boldsymbol{b}) = O_{j-l,k-m}(\boldsymbol{b}) \ .$$

The local to local (L2L) operator can be used to shift a local expansion from its center to another position inside the expansion sphere (see Figure 2.6). This means, a local expansion $\mu(\boldsymbol{r})$ around the center $\boldsymbol{r}$ can be shifted to a local expansion $\mu(\boldsymbol{r}-\boldsymbol{b})$. The coefficients of the new local expansion can be computed using:

$$\mu_{lm}(\boldsymbol{r}-\boldsymbol{b}) = \sum_{j=l}^{p}\sum_{k=-j}^{j} C_{jk}^{lm}(\boldsymbol{b})\, \mu_{jk}(\boldsymbol{r}) \ . \tag{2.7}$$

Similar to the M2M operation, the re-expansion of the new local expansion does not require to use the original particles. In contrast to M2M this operator introduces additional errors with respect to the expansion length. As seen in Equation 2.7 the computation of the new local expansion has a complexity of $\mathcal{O}(p^4)$.

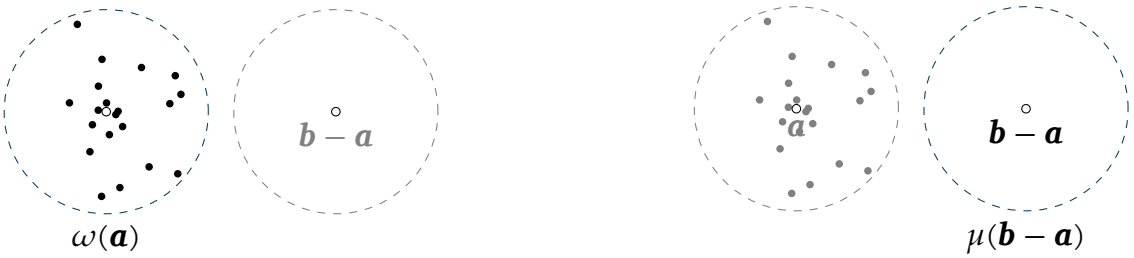


Figure 2.7: The multipole to local operator translates a multipole expansion at a center $\boldsymbol{a}$ to a local expansion at $\boldsymbol{b}-\boldsymbol{a}$ outside of the sphere of the multipole expansion.

2.3.3 Multipole to Local

Definition 2.10 *The **multipole to local** operator B_{jk}^{lm} is given by:*

$$B_{jk}^{lm}\left(\boldsymbol{b}\right) = M_{j+l,k+m}\left(\boldsymbol{b}\right) .$$

The multipole to local (M2L) operator can be used to translate a multipole $\omega(\boldsymbol{a})$ into a local expansion $\mu(\boldsymbol{b}-\boldsymbol{a})$ (see Figure 2.7). Assuming a multipole expansion for particles in a sphere around the center $\boldsymbol{a}$, this multipole represents the impact of the expanded particles on any point outside the sphere. Using the M2L operator a local expansion at a point $\boldsymbol{b}-\boldsymbol{a}$ outside the sphere around $\boldsymbol{a}$ can be computed. This local expansion represents the impact of these particles on points inside the sphere of the new local expansion. For the radius of the new local expansions it is required, that the two spheres are disjoint. This translation can be computed without using the particles as proposed in Equation 2.4. The computation can be done using the following formula:

$$\mu_{lm}\left(\boldsymbol{b}-\boldsymbol{a}\right) = \sum_{j=0}^{\infty}\sum_{k=-j}^{j} M_{j+l,k+m}\left(\boldsymbol{b}\right)\omega_{jk}\left(\boldsymbol{a}\right) . \tag{2.8}$$

This operator is the only operator introducing an additional operator error since the summation of j is limited to p. Nevertheless, this error can be estimated a priori and controlled [69] within the FMM. Equation 2.8 shows that the application of this operator exhibits $\mathcal{O}(p^4)$ complexity. Now, all operators required for an FMM have been derived.

2.4 FMM Algorithmic Flow

The computation of forces and potentials using the FMM is split up into far-field and near-field computations. For $p \to \infty$ the forces $\boldsymbol{F}_{FF}$ and potentials Φ_{FF} are exact. The forces $\boldsymbol{F}_{NF}$ and potentials Φ_{NF} are always exact since no expansions will be used in the near-field. For the sake of simplicity the error introduced from numerical round-off errors will be ignored. Since the forces and the potentials are cumulative they can be written as:

$$\boldsymbol{F} = \boldsymbol{F}_{NF} + \boldsymbol{F}_{FF}$$

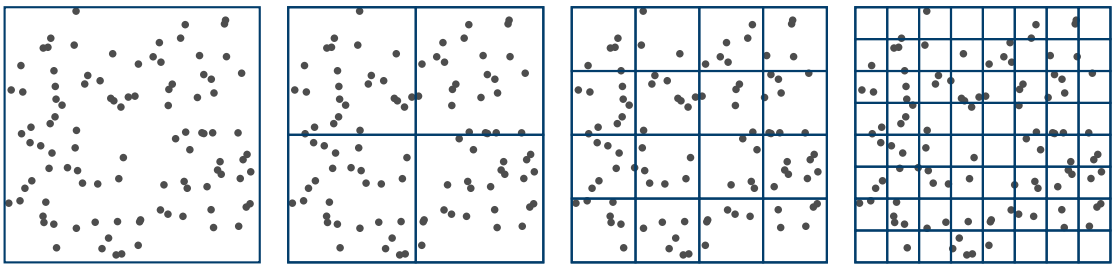


Figure 2.8: An example of a quadtree for subdivision depth zero, one, two and three.

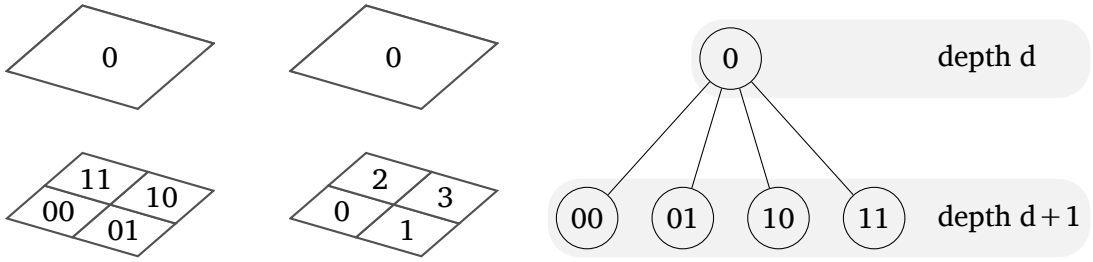


Figure 2.9: The spatial representation of a quadtree. In the first figure the nodes are indexed using a two dimensional index. In the second figure the nodes are indexed with a one dimensional index using a space filling curve. The last figure shows the corresponding graph representation of the quadtree.

and

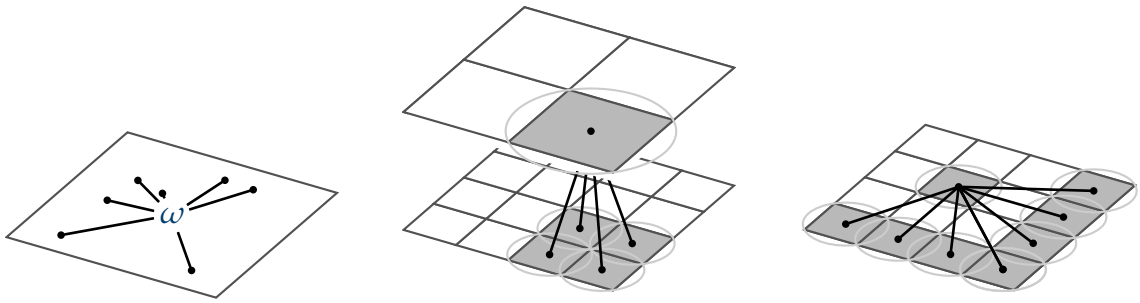
$$\Phi = \Phi_{NF} + \Phi_{FF}\,.$$

2.4.1 FMM Setup

Spheres are ideal to describe regions of convergence, but unfortunately they are not well suited to fill space without overlapping. For the algorithm, a space filling shape without overlap is required. Therefore, the algorithm used in this work, utilizes cubes. Cubes can be handled easily and are filling space. Other shapes are possible, but complicate the algorithm and convergence requirements. To keep things simple, only cubes will be discussed here. In the following, cubes are also called boxes.

Spatial Subdivision using Octree

Since all of the introduced operators require a certain spatial subdivision, the FMM has to provide this subdivision as a first step of the algorithm. The FMM uses an octree to divide space. In an octree, every vertex, except the leaf vertices has 8 child vertices. Even though the algorithm works in 3D using octrees, the shown examples are for simplicity only in 2D (see Figure 2.8). The creation of the octree can be setup as follow: It starts with a simulation box which represents the root vertex (see Figure 2.9). This simulation box is then subdivided recursively until a certain maximum depth $d_{\max}$ is reached. This maximum depth must be defined by the user or in this implementation it is set automatically by the error and runtime optimizer. Every level in the tree has 8^d vertices, where d denotes



(a) P2M (b) M2M (c) M2L

Figure 2.10: The particle to multipole (P2M) operation expands the particles at the center of a box on the lowest level in the tree in a multipole expansion ω. The multipole to multipole (M2M) operation shifts the multipoles from one center of a child box to the center of the parent. This step is repeated for all child boxes of one parent box. The expansions are accumulated at the parent box. The multipole to local (M2L) operation translates multipole expansions to local expansions. Expansions with the same center can be summed up.

the depth of the subdivision. After the last subdivision, the boxes on the lowest level, the leaf vertices are reached. The leaves of the tree are the smallest boxes in the spatial tree each covering $1/8^d$ of space.

Binning Particles

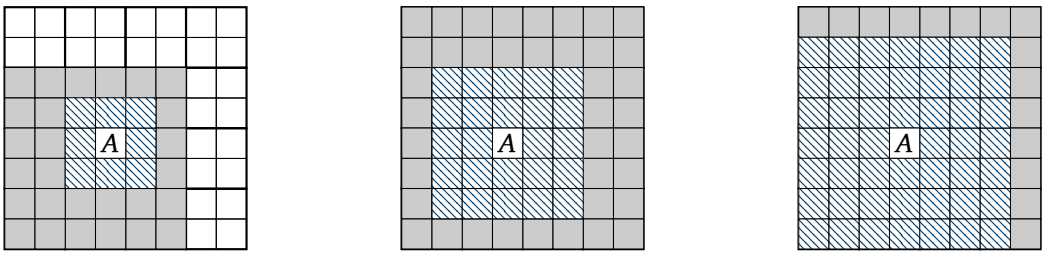
The presented octree has subdivided the simulation space so far. However since the computed properties stem from particles a radix sort [27] is used to bin the particles into the boxes on the lowest level.

2.4.2 Far-Field Computation

For the far-field computation the particles need to be expanded into multipole expansions. For this, one multipole for each box on the lowest level will be expanded (see Figure 2.10a) at the center of the box using only particles contained in the box. As discussed before, the expansion of one particle has the computational complexity of $\mathcal{O}(p^2)$ and is done for all N particles, hence this step has $\mathcal{O}(Np^2)$ complexity.

After all multipole expansions for all boxes on the lowest level are computed, the multipoles need to be distributed in the tree at higher levels starting from $d = d_{\text{max}} - 1$ to $d = 0$. This is achieved by using the aforementioned M2M operator. As shown in Figure 2.10b the multipoles are shifted from the center of the box to the center of the parent box. Since every parent box has eight child boxes, this results in eight shifted multipole expansions for each parent box. These multipoles can be accumulated since they share the same center.

After the M2M step, all boxes in the tree own a multipole expansion representing the particles contained therein. For further computation, the multipoles must be translated into local expansions using the M2L operator. For this operator it



(a) $ws = 1$ (b) $ws = 2$ (c) $ws = 3$

Figure 2.11: Three examples of the M2L interaction set using different well separation criteria in 2D.

is necessary, that the surrounding spheres of the multipole expansion and the local expansion to be generated do not overlap. However, the sphere of direct box neighbors in the octree overlap. The requirement is satisfied by the well separation criteria *ws* and the thereby defined interaction set.

The well separation criteria *ws* is a FMM parameter usually provided by the user. The interaction set is defined as follows:

1. Let us consider any box A inside the tree.
2. Only *ws*-neighbors of A's parent box are considered in the interaction set of the box A.
3. All child boxes of these boxes under consideration are added to the interaction set.
4. Afterwards, all *ws* direct neighbors of A itself are excluded from the interaction set again.

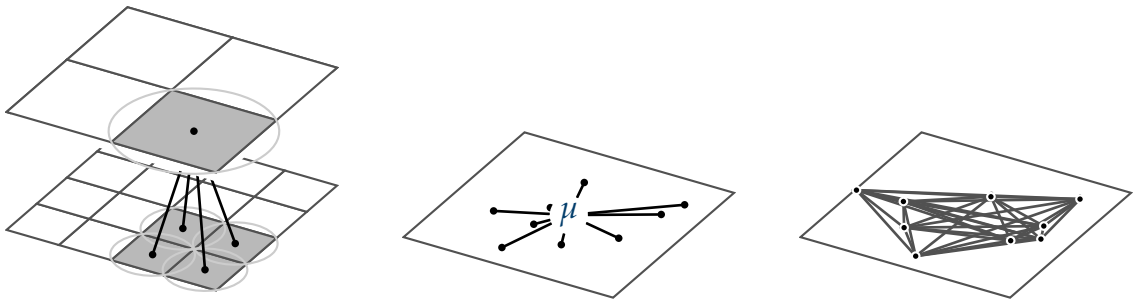
This results in 189 interacting boxes for $ws = 1$. In general, the maximum number of boxes in the interaction set depending on *ws* is given by:

$$(2\,(2ws+1))^3 - (2ws+1)^3\,.$$

An example of the size of interaction sets can be seen in Figure 2.11.

This means, $ws = 1$ defines the smallest distance satisfying the non-overlapping criteria. A large well separation criteria increases the distance between the target box and the boxes in the interaction set. After the interaction set is defined, the M2L operator (see Figure 2.10c) is applied for each box in the interaction set and computes a local expansion from all multipole expansions of boxes in the interaction set at the center of box A.

After the M2L step, local expansions representing the potentials and forces of particles in the interaction set are scattered within the full tree. To maintain the linear complexity of the algorithm all local expansions from higher tree levels need to be shifted to their leaf nodes. This is done using the L2L operator. As seen



(a) L2L (b) L2P (c) P2P

Figure 2.12: The local to local (L2L) operation shifts the local moments from the center of the parent box towards the center of all child boxes. The local to particle (L2P) operation computes the far-field forces influencing the particles in the box. The particle to particle (P2P) operation computes the near-field interactions using a direct solver.

in Figure 2.12a the operator is used to shift the local expansion from a parent box to all child boxes. This is done starting at the root node on downwards. Since every child box has already a local expansion from the M2L step on its respective level, the shifted local expansion and the local expansion from the M2L step have to be accumulated. After the lowest level in the octree containing the leaf nodes is reached, the resulting local expansion represents all particles in the far-field of this box. These local expansions on the lowest level will be used to compute the far-field forces and potentials affecting the particles using the L2P operator.

Definition 2.11 *The potential Φ_{FF} and force F_{FF} for a particle at position $\boldsymbol{a}_k = (a_k, \alpha_k, \beta_k)$ in a box with the corresponding local expansion $\mu_{lm}(\boldsymbol{r})$ can be computed using the* ***local to particle*** *operator with the following equations:*

$$\Phi_{FF}(\boldsymbol{a}_k) \approx \sum_{l=0}^{p} \sum_{m=-l}^{l} \mu_{lm}(\boldsymbol{r}) \frac{a_k^l}{(l+m)!} P_{lm}(\cos\alpha_k)\, e^{-im\beta_k}$$

$$\boldsymbol{F}_{FF}(\boldsymbol{a}_k) \approx \sum_{l=0}^{p} \sum_{m=-l}^{l} \mu_{lm}(\boldsymbol{r}) \nabla_{\boldsymbol{a}_k} \left[\frac{a_k^l}{(l+m)!} P_{lm}(\cos\alpha_k)\, e^{-im\beta_k} \right]$$

As seen in the equations the computation of the force and the potential has a computational complexity of $\mathcal{O}(p^2)$. Similar to the P2M step this step is performed for all N particles resulting in a total complexity of $\mathcal{O}(Np^2)$ for this step.

2.4.3 Near-Field Computation

The near-field forces and potentials have not been taken into account yet. These are computed from M particles contained in the *ws* neighboring boxes. The computation itself is only done with this limited set of particles and Equation 2.1 and Equation 2.2, exhibiting a computational complexity of $\mathcal{O}(M^2)$. It is done for

N particles and thus results in a total complexity of $\mathcal{O}(M^2N)$. Since the number of particles can be fixed to M for any given system, the total complexity of the FMM is still $\mathcal{O}(N)$.

2.5 FMSOLVR – IMPLEMENTATION SPECIFICS

Fmsolvr [41] is an FMM implementation specialized for MD simulation developed at the Jülich Supercomputing Centre (JSC). It is part of the SPPEXA [21] project GROMEX [53]. Fmsolvr is implemented using modern C++11 and relies heavily on language and standard library features like vectors or template-meta-programming. In the following, implementation specifics important for this work are presented.

As a brief summary, the complete algorithm flow is shown in Figure 2.13. In the *Fmsolvr* implementation the sequential flow can be described in five distinct passes.

Pass 1 computes P2M and M2M.

Pass 2 computes all M2L steps on all levels.

Pass 3 includes all L2L operations.

Pass 4 computes the far-field forces and potentials via the L2P operator.

Pass 5 computes the near-field forces and potentials via P2P.

The operators (M2M, M2L and L2L) presented before, exhibit a computational complexity of $\mathcal{O}(p^4)$. Using an additional rotation operators, it is possible to reduce the complexity to $\mathcal{O}(p^3)$ [117]. Nevertheless, for the parallelization proposed in this work, it is not important, which kernel implementation is used. This is a decision done by the algorithm developer and is fully independent from the task engine proposed in the following chapters.

The most important data structures for the algorithm are the multipole expansion and the local expansion. As discussed before, these expansions exhibit a triangular structure. Therefore, the data structure used to store the expansion coefficients has a triangular structure as well using generic complex floating point types for the coefficients itself. It can be configured to store the coefficients column or row major.

Furthermore, the multipole expansion and local expansion have an not yet discussed symmetry. The part for $m \geq 0$ of the coefficients in the triangular

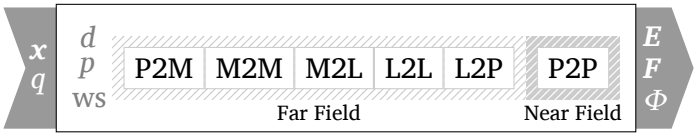


Figure 2.13: The algorithmic flow of the sequential FMM. The parameter d, p and ws are required to set the accuracy of the algorithm.

structure can be used to compute the part for $m < 0$ of the triangular using the following relation:

$$\omega_{l,-m} = (-1)^m \cdot \overline{\omega}_{l,m}$$
$$\mu_{l,-m} = (-1)^m \cdot \overline{\mu}_{l,m}$$

This reduces the memory required for the expansions and can be beneficial for data transfer operations. The implementation can be configured at compile time to store the complete coefficient triangular or only the upper half. Besides the expansions, the implementation has custom data structures for particles and their corresponding charges.

Chapter 3

A C++ Task Engine

Shared memory parallelization becomes more and more important in the age of exascale, single core consumer hardware barely exists anymore. Even today's smartphones already feature octa-core processors. To make things worse those processors are heterogeneous which means, that the cores have different capabilities like different clock speeds. For both, smartphone hardware and HPC hardware, the number of cores per node increases rapidly. By extrapolation, 1000 cores per node can easily be reached within the next few years. This development poses new challenges for the software development, especially in HPC.

Available strategies for parallelizing HPC nodes with a moderate number of cores are not necessarily adaptable for thousands of cores. Additionally, the increasing number of cores is not the only development impeding the exploitation of performance on a single node. Things like single instruction, multiple data (SIMD), heterogeneous cores, caches or hierarchical memory need to be considered as well. How could an increase of complexity be coped from a software developer point of view? The concept of abstraction allows to map certain hardware features to the software layer in a generic manner.

To understand this problem from the viewpoint of a software engineer the hardware developments need to be distinguished into two different categories.

Qualitative changes in hardware describe the introduction of new features, not present in one hardware generation, but available in the next, like the multi-core processors or vectorization (SIMD).

Quantitative changes describe the enhancement of existing features like the increasing number of cores in a multi-core processor from only a few to several dozens.

Obviously, it is not possible to foresee qualitative changes and the software cannot adapt to them automatically. However, to some degree it is possible to account for quantitative changes. To cope with these challenges, primarily one thing is required from modern software engineering: software abstractions of hardware features. With abstractions, quantitative changes are already represented in the

software and the achieved performance then only depends on the quality of the implementation.

For HPC applications, especially latency-critical ones, strategies that are capable to map performance critical hardware features into software abstractions are vital already today. In this chapter a task engine for HPC is presented. The task engine design especially focuses on latency-critical applications.

The proposed task engine will be introduced for intra-node usage in this chapter. Since the tasking approach in general is a relatively new one in HPC the chapter starts with a discussion of current intra-node parallelization approaches and their limitations.

3.1 State of the Art Parallelization Approaches

Historically, intra-node parallelization was not part of any programming language like C or Fortran, since supercomputers started out with only one core per node and hence no necessity to parallelize on the node. Therefore, language extension or external libraries have been used for the parallelization in the early days of multicore nodes.

For some HPC application on today's hardware it might even be sufficient to solely use inter-node parallelization using the message passing interface (MPI) on the node. Those HPC applications use multiple MPI processes on a single node. Since, inter-node parallelization was required on supercomputers anyway, this is a working solution without additional efforts. Nevertheless, unnecessary additional overhead is introduced by this approach. The overhead stems from an increased memory foot print due to usage of additional MPI processes instead of lightweight threads. Also, data must be explicitly communicated instead of directly accessed in shared memory by threads resulting in unnecessary duplication of data and communication overhead. This approach lacks any usage of shared resources and especially gives up the advantage of using the shared memory.

For only few cores per node, this is not a problem and may be acceptable to ignore the overhead of additional MPI processes. But for future supercomputers it will not be possible to gain performance with such an approach anymore.

3.1.1 Loop-Level Parallelization

Loop-level parallelism is one of the most common approaches for intra-node parallelization. The main idea of this approach is to obtain parallelism from existing loops (e.g. for-loop). The independent loop iterations are split up and parallelized by assigning subsets of the iterations to different threads.

Without inter-loop data-dependencies or other synchronization requirements and a sufficient number of iterations, this approach scales efficiently for a single loop. However, applications usually encompass more then a single loop and exhibit different inter-loop dependencies. Additionally, due to remaining sequential parts outside the loops the scaling is limited quickly by Amdahl's law [5]. Amdahl's law

gives an upper bound for the achievable speedup S depending on the sequential portion r_s and parallel portion r_p of the program executed on P cores as

$$S(P) = \frac{1}{r_s + \frac{r_p}{P}} \quad (r_s + r_p = 1). \tag{3.1}$$

The limit value of Amdahl's law shows the upper bound of the speedup depending on the sequential parts of the program:

$$\lim_{P \to \infty} S(P) = \frac{1}{r_s}. \tag{3.2}$$

Those sequential regions may stem from different sources:

- Data-dependencies or other synchronization requirements within the algorithm will sequentialize the program.
- Computations outside of parallelized loops will remain sequential.
- Multiple subsequent parallel loops may utilize thread creation via fork-join which requires synchronization at the end of the loop and thereby introduces sequential parts at the synchronization points.

To give an example, let's assume a program has a sequential portion of 5 % the achievable speedup is limited by 20. On a machine encompassing 40 cores, this would lead to a maximum parallel efficiency of 50 %. This utilization is completely insufficient. The limitation in scaling due to the sequential regions might only be acceptable for low core numbers, but unacceptable for future systems.

Furthermore, loop-level parallelism is limited by the distribution of work. It might be the case, that not all loop iterations encompass the same amount of work. This leads to a load imbalance which has to be actively resolved introducing additional overhead.

The main drawback of loop-level parallelism however stems from the concept itself. The algorithm developer is forced into an artificial loop-based view of his algorithm. Of course it makes sense to parallelize existing loops with this approach. But, it is counter intuitive to introduce new arbitrary loops for the sole purpose of parallelization. It seems to be much more intuitive to think about task-based parallelization, which inherently avoids any predefined structure of parallel execution.

Pragmas and OpenMP

The de-facto standard to exploit loop-level parallelism is OpenMP [28]. OpenMP is an application program interface (API) for intra-node parallelization. It was first published in 1997 by the OpenMP architecture review board focusing on loop-level parallelism. Starting with version 3.0, OpenMP introduced task-based functionalities. OpenMP natively supports Fortran, C and C++ and implementations are available for almost all common compilers used on today's supercomputers.

OpenMP is based on compiler directives, so called pragmas. Pragmas provide an easy-to-use syntax for parallelization, especially for languages like Fortran and C. Those pragmas are being substituted during the compile process by the OpenMP compiler itself.

Pragmas are language extensions and therefore are not part of the programming language standardization process. Especially from the view point of modern C++, these kind of extensions seem to be outdated. C++ offers various zero-cost abstraction features and parallelization functionality as part of the language standard. Thus, additional compiler directives are not required to handle intra-node parallelization.

Additionally, OpenMP lacks sophisticated scheduling for latency-critical applications which may require the prioritization of a critical path. For example, the FMM exhibits a parallelism bottleneck in the upper tree levels. With OpenMP it is not trivially possible to influence the scheduling in such a way that the prioritization of the critical path can take place.

Furthermore, it is not possible as a user to influence the program flow inside the OpenMP library. Since it is not desirable to change the compiler implementation of OpenMP, it is not possible to change any program specifics that might be required to enable certain optimizations. In contrast, for a C++ library it would be easier to exchange components and adapt for different scenarios.

3.1.2 Task-Based Parallelization

Instead of exploiting parallelism solely from loops, task-based parallelization splits up the entire program flow into several units of work and their dependencies. These units of work are called tasks and may be executed in parallel. For a task-based parallelization, dependencies need to be tracked in order to maintain the correct execution of the tasks.

Due to the increasing number of cores and the limitation of loop-level parallelism, task-based parallelization becomes an increasingly popular approach in HPC. There exit several libraries implementing different APIs for task-based parallelization like Argobots [102], Intel TBB [96], StarPU [9], OpenMP Tasks [10] or Wool [39]. Due to the availability of many different task engines, only the most prominent ones are discussed shortly.

Argobots

Argobots [102] is a lightweight task engine written in C. The focus of Argobots is to offer a lightweight task engine while overcoming scaling bottlenecks due to blocking function calls like MPI communication. To tackle those blocking calls, Argobots uses so called user-level-threads (ULT). ULT can yield back to the scheduler of Argobots during their execution. This is done by copying and switching the current state of execution. This behavior can also be found in Boost context library [17]. The scheduler reschedules the partially executed task at a later point in time. This can be valuable while waiting for a lock, an expensive I/O operation or inter-node communication.

Argobots also offers a basic work-stealing scheduler as well as LIFO, FIFO and bucket-based priority queues. Furthermore, Argobots provides the concept of stackable and nested schedulers. This means, users can define their own schedulers and nest them for more sophisticated scheduling.

Finally, the relation to the MPICH development and the coupling between MPICH and Argobots should be mentioned. The developer propose an Argobots-aware MPICH implementation [98], supporting ULTs. This may lower the overhead of multithreaded communication.

Intel TBB

Intel thread building blocks (TBB) is a versatile task engine written in C++. It reflects the effort of Intel providing tools for programming especially highly scalable processors like Intel Xeon Phi [62] or Intel Xeon Scalable [65] processors.

Intel TBB offers tools to support loop-level parallelism as well as task-based parallelism. Instead of using compiler directives and pragmas, Intel TBB offers program interfaces written in modern C++. Intel TBB is based on the threads of the C++ standard library.

The task engine also encompasses two different data structures for managing dependencies [61]. The first is a dependency graph implementation and the second a data-flow graph implementation. For both graphs the user needs to create a corresponding graph object. Afterwards, the edges and vertices representing the algorithmic flow need to be defined. The actual dispatching of tasks is done at runtime.

OpenMP Tasks

OpenMP is an API for shared memory computing. It is available for C, Fortran and C++ as a compiler extension. Since version 3.0 OpenMP also offers task-based parallelization features. However, OpenMP is entirely based on compiler directives called pragmas.

For the implementation of task-based parallelism OpenMP focuses on so called divide and conquer strategies [91]. The user can define task regions. Inside these regions it is possible to recursively define other tasks as siblings. All tasks are created from those regions and handed over to the OpenMP runtime system. The tasks can be joined using a barrier either inside a task for all sub-tasks or globally. As long as no dependencies or data synchronizations are required, the execution order of the task is unknown to the user.

Benchmarks show, that this approach does not scale appropriately [97, 20].

OpenMP extended the tasking with dependency resolving. It allows the user to define dependencies for a single task. This is also done behind the scene and no further details are adjustable by the user.

3.1.3 Modeling Algorithmic Dependencies

For a more fine-grained task-parallel execution of an algorithm, the algorithmic dependencies need to be modeled. There are two common approaches for this. The first is the dependency graph which models the algorithm in a backward view.

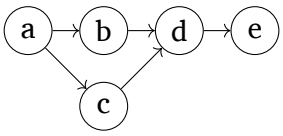


Figure 3.1: Graph drawing example of the graph $G = (V, E)$ with $V = \{a, b, c, d, e\}$ and $E = \{(a, b), (a, c), (b, d), (c, d), (d, e)\}$.

The second is the data-flow graph modeling the dependencies in a forward view. Both use the mathematical structure of a graph.

Definition 3.1 *A **graph** is a pair $G = (V, E)$ consisting of a set of* vertices *V and* edges *E.*

Edges are ordered or unordered pairs of elements of V. Iff the edges are ordered pairs, the graph is called *directed*. Iff the edges are unordered pairs, the graph is called *undirected*. A vertex $v \in V$ and an edge $e \in E$ are incident, iff $v \in e$. Two vertices are called adjacent, iff they incident a common edge. Two edges are called incident, iff they are incident with a common vertex.

Definition 3.2 *A **walk** is defined as a sequence of vertices $(v_0, \ldots, v_n)$ with $v_i \in V; n \in \mathbb{N}; n \geq 1$. Vertices v_i and v_{i+1} are adjacent $\forall i \in \mathbb{N}; i \leq (n-1)$.*

Definition 3.3 *A **closed walk** or a **cycle** is a walk with $v_0 = v_n$.*

Definition 3.4 *An **acyclic graph** is a graph without cycles. A directed acyclic graph (DAG) is a directed graph without cycles.*

A graph drawing is the geometric representation of a graph (see Figure 3.1). This representation however is not unique.

Dependency Graph

Unfortunately, there is no uniform definition of a dependency graph in literature. In this work a DAG is called a *dependency graph*, if it represents the dependencies of every piece of data. This means, every vertex represent one and only one piece of data and every edge represents one and only one manipulation of it.

Insights into the parallelization can be obtained by the structure of such a dependency graph. There exist several scientific publications showing how to partition those graphs and distribute the work accordingly. Nevertheless, those approaches are limited by the complexity of the optimal partitioning being NP-complete [60]. This might introduce a significant overhead for latency-critical applications.

For the parallel execution of an algorithm, the dependencies need to be fulfilled before the next task can be executed. This can be checked using the dependency graph. The drawback of this approach is, that it requires the setup and traversal of the complete graph within the program. Even if it is not required to store the complete graph in memory, this introduces runtime overhead.

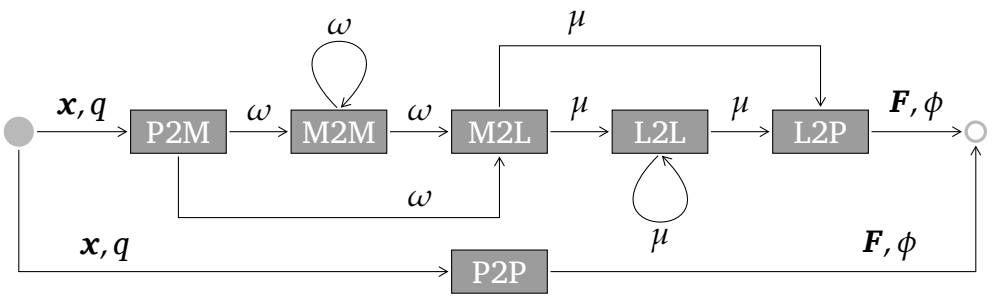


Figure 3.2: An example of a data-flow graph. This data-flow graph denotes the dependencies of the FMM. The nodes represent FMM specific operators. The edges denote the data-dependency between the operations.

Additionally, it requires to create a significant number of tasks upfront. Afterwards, the task engine needs to iterate over all created tasks and check if their dependencies are met. This causes a constant polling on the created tasks for checking the dependencies without computation.

Data-Flow Graph

In contrast to the dependency graph, the data-flow graph does not model every single piece of data. The data-flow graph is data-centric and only models pipelines of data manipulations. In the data-flow graph a vertex denotes an abstract operation in the algorithm, whereas an edge denotes the input and output data of an operation. It is important to mention, that edges and vertices only represent abstract data and operations and not a concrete piece of data or operation. The start node of the graph is used for the algorithms input data. The input data is used and manipulated by the following operations in the graph until the end of the graph is reached. The last node in the graph represents the output data of the algorithm.

Figure 3.2 shows an example of a data-flow graph used for the FMM. All operations are represented by a single node in the graph and their corresponding input and output data is denoted on the edges. As seen in this example, the data-flow graph may contain cycles and is therefore not acyclic.

3.2 Task Engine Design Fundamentals

In this section the design and implementation of the proposed task engine will be discussed in detail.

In the following, the components shown in Figure 3.3 are described from the bottom to the top:

Thread On every physical core of the CPU a thread will be created. This thread can be created by any available system thread implementation (e.g. pthreads). Since the task engine is written in C++11 this is almost always an `std::thread`.

ThreadingWrapper Since the implementation must be independent from the actual system threads, the `ThreadingWrapper` is used for encapsulation. It

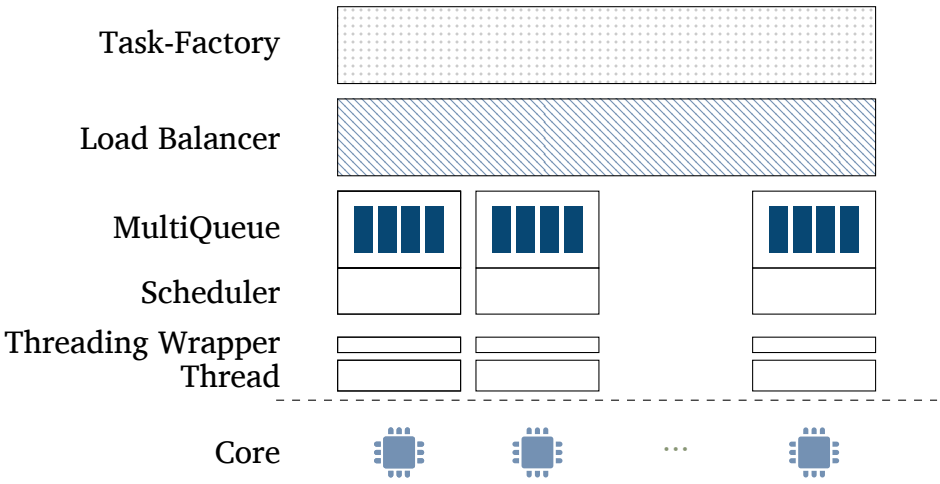


Figure 3.3: An overview of the main infrastructure of the task engine.

maps the interface of the thread to the task engine. This wrapper encompasses a constructor for the creation of a new thread as well as a method for retrieving the unique identifier of a thread. Since this is depending on the used system threads, the wrapper needs to be specialized correspondingly.

Scheduler The scheduler defines the order of execution of tasks in a single thread. The scheduler will consist of different components (e.g. the work-stealing scheduler or the type-driven priority scheduler).

MultiQueue Tasks need to be put in a queue while waiting for being executed. The multi-queue is an extension of a priority queue. It is used for the type-driven priority scheduling of tasks along a critical path. The multi-queue is private to its corresponding thread. However, for the purpose of work-stealing or work-sharing it can be shared among other threads.

LoadBalancer When tasks are created, a certain thread needs to be selected for the execution. The load balancer distributes the newly generated tasks between the existing multi-queues and thereby between the threads. This load balancer is used by all threads and handles task distribution. Specific load-balancing strategies must be defined by the user.

TaskFactory The task factory combines all components required for the creation of new tasks. Every task is able to access the task factory for the creation of new tasks. Additionally, the task factory contains the static data-flow dispatcher.

Suspending Threads

HPC applications may consist of several modules or libraries. Not all parts of a program may use the proposed task engine for the intra-node parallelization. In this case, it is necessary to release the used resources (e.g. threads) to make them

LISTING 3.1: THREADDORMITORY IMPLEMENTATION

```
1  class ThreadDormitory {
2    void trySleep() {
3      if (soft_sleep) {
4        std::unique_lock<std::mutex> unique_lock_guard(mutex);
5        cv.wait(unique_lock_guard, [&]() { return !soft_sleep; });
6      }
7    }
8    void putToSleepAll() {
9      std::lock_guard<std::mutex> lock(mutex);
10     soft_sleep = true;
11   }
12   void wakeUpAll() {
13     {
14       std::lock_guard<std::mutex> lock(mutex);
15       soft_sleep = false;
16     }
17     cv.notify_all();
18   }
19   bool soft_sleep = false;
20   std::mutex mutex;
21   std::condition_variable cv;
22 };
```

available to other parts of the program. Usually this is done by joining the threads after the computation. If this is done repeatedly, a significant overhead will be introduced especially for latency-critical applications.

Instead of forking and joining threads over and over again, task engine suspends the threads. This is done with the help of condition variables. During the actual suspend, the threads will wait on this condition variable. Since this waiting does not use busy loops, the resources can be used by other parts of the program in the meantime. Besides this, suspending introduces less overhead then forking and joining threads. This capabilities are implemented in the `ThreadDormitory`.

The implementation of the `ThreadDormitory` is shown in Listing 3.1. The implementation only uses a condition variable and mutex lock from the C++ standard library. The `trySleep` method is similar to the reference implementation for using condition variables [26]. The wait method of the condition variable requires a unique lock which will be released automatically inside the wait method itself. Additionally, a lambda is used to prevent spurious wake-ups.

Thus the suspend works as follows: At the beginning the `soft_sleep` flag is initialized to false. All threads are created and start the execution of available tasks. After the execution of a task is finished, each thread checks the flag using the `trySleep` method. Since the method can be inlined by the compiler there is no overhead for this function call. When the computation is completed, the main thread calls the `putToSleepAll` method. This sets the flag `soft_sleep` to true. Whenever a thread calls the `trySleep` method again, it will be suspended using the condition variable.

For the start of the next computation phase using the task engine, the main thread calls the `wakeUpAll` method. This sets the sleep flag to false and notifies all

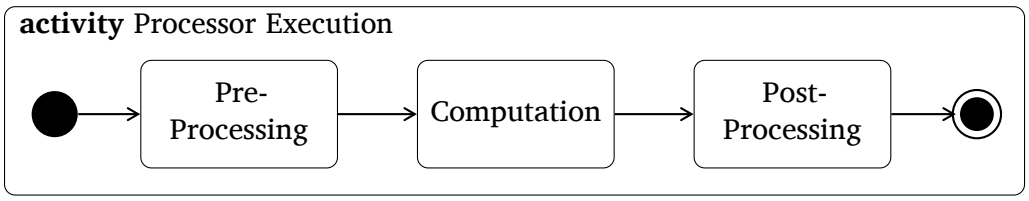


Figure 3.4: The UML activity diagram of the three-phase execution of a processor.

sleeping threads via the condition variable. Afterwards, the threads can be used by the task engine as before.

3.2.1 Anatomy of a Task

Until now, the task for the task engine are missing. The tasks used in the task engine consist of two parts. The first is a set of unique identifiers and the second is a reference to a processor object. The set of unique identifiers is used for the identification of data objects manipulated by the task. For the FMM these identifiers refer to boxes in the tree. The processor object is similar to a stateful function callback and wraps the functionality of a task. The actual execution of a task is the execution of the processor by applying the identifiers.

Three-Phase Execution

For some applications the offloading of data to an accelerator prior the actual computation of a task or the reducing of data after the computation of a task may be required. To support this, the execution of a processor is split into three phases (see Figure 3.4). This three-phase execution consists of a pre-processing method, a computation method and a post-processing method. All methods share the same execution context and are executed sequentially, one after another.

Supporting User-Defined Processors

A processor is the main component of a task and defines its functionality. Since tasks are usually defined by the user, it must be easy to implement user-defined processors. This means, the user should not be required to repeatedly implement the default behavior (e.g. three-phase execution). This could be solved by virtual inheritance, but this would limit the granularity of the tasks unfavorable.

The processor class is implemented using an abstract processor and the aforementioned CRTP feature (compile-time inheritance). Parts of the abstract processor are shown in Listing 3.2.

The abstract processor has one template parameter reflecting the concrete processor. This template parameter is a so called template-template parameter. A template-template parameter can be used to apply a template name instead of a concrete type like `std::vector` instead of `std::vector<double>` as a template parameter. In this case, `std::vector` is not a concrete type and could not be applied for a normal template parameter. For the processor this is used to allow additional template parameters for the concrete processor. To retrieve the processor type, these template parameters need to be applied as done in the type definition at

LISTING 3.2: ABSTRACTPROCESSOR CLASS IMPLEMENTATION USING CRTP

```
1  template <template <typename...> class ConcreteProcessor, [...]>
2  struct AbstractProcessor {
3    using Processor = ConcreteProcessor<[...]>;
4    [...]
5    void execute(BoxIDType box_id) {
6      concrete_processor().pre_processing(box_id);
7      concrete_processor().run_computation(box_id);
8      concrete_processor().post_processing(box_id);
9    }
10   /**
11    * All method should be empty by default and hidden with an implementation in
12    * child class/concrete processor if needed.
13    */
14   void pre_processing(BoxIDType) {}
15   void run_computation(BoxIDType) {}
16   void post_processing(BoxIDType) {}
17   [...]
18   Processor & concrete_processor() {
19     return *static_cast<Processor *>(this);
20   }
21 };
```

LISTING 3.3: P2MPROCESSOR AS A CONCRETE PROCESSOR EXAMPLE

```
1  template <typename... Args>
2  struct P2MProcessor : public AbstractProcessor<P2MProcessor, Args...> {
3      using P2MProcessor::AbstractProcessor::AbstractProcessor;
4
5    void run_computation(BoxIDType box_id) {
6      fmsolvr::pass1_P2M(
7          this->fmm_handle_, box_id, this->task_factory_, this->dm_);
8    }
9  };
```

line 3.

The `execute` method at line 5 defines the three-phase execution of the processor by calling the three defined methods. Those three methods are implemented empty from line 14 to line 16. Additionally, the type casting method called `concrete_processor` is used. This methods converts the `this` pointer to a pointer to the concrete processor. This can be done with a static cast and is thereby converted at compile-time. All methods, especially the three methods `pre_processing`, `run_computation` and `post_processing` can be implemented in the derived class. This will hide the implementation in the abstract class and ensures, that the implementation of the concrete processor is executed.

Listing 3.3 shows the implementation of the P2M specific processor. Since this processors does not require any pre- or post-processing, only the `run_computation` method is implemented at line 5. Additionally, the abstract processor offers a constructor which is inherited at line 3. This is the only requirement for implementing a new processor.

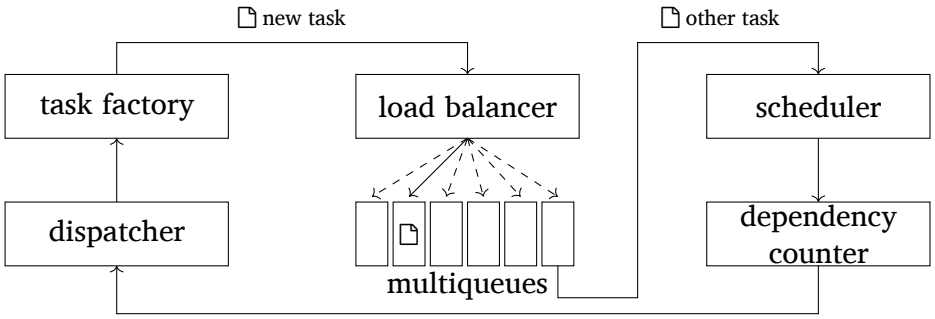


Figure 3.5: The life-cycle of a task inside the task engine. The scheduler will execute a task from a multi-queue. After the execution of the task, the corresponding dependency counters are decremented. If a dependency counter becomes zero, the dispatcher will be invoked. If the dispatch results in the creation of a new task, the task factory is employed. The newly created task will be handed over to the load balancer. The load balancer decides which multi-queue on which thread is used for enqueuing. Afterwards, the cycle starts over again. This cycle happens in parallel on all threads.

Ranges and Vectorization

Vectorization becomes more and more important for modern HPC applications. Enabling vectorization is also a challenge for task engines using fine-grained tasks. Since the work is split up into very small chunks, those small tasks might not exhibit enough vectorization capabilities themselves. Also, the data used in a single task might not necessarily fit the specific vectorization width.

The efforts of vectorization itself go beyond the scope of this work, but the challenges for the task engine must be addressed. It is required from a task engine to provide tasks with an adjustable amount of work. To enable this flexibility, the task engine has to be extended in the following way: The task class described before handles only a single identifier as a parameter. This parameter must be exchanged by a set of identifiers. This allows the task engine to adjust the actual computational size of a task depending on the size of this set. Instead of computing a task for a single ID, the new task resembles work required for all IDs inside the new set. Additionally, the task engine must be able to split up those identifier sets again and fuse two sets to a new set. This enables the task engine to adjust the amount of work performed by a single task by merging or splitting the identifier sets.

This feature is used to support vectorization within the task engine. For retrieving the next task from the queue, the user can set the desired amount of work which should be included in the task. The queue itself may contain tasks with an arbitrary amount of work. During the retrieving the queue will merge and split up tasks until the preferred size is reached. With this feature, the user can adjust the amount of work of the task pulled out of the queue.

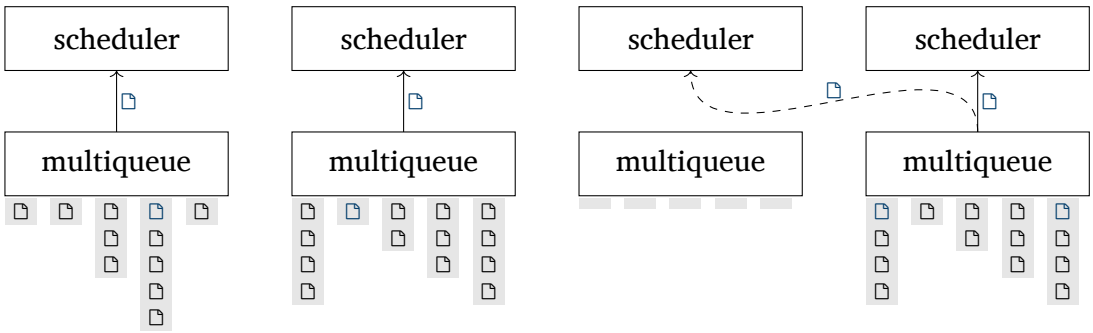


Figure 3.6: An example of the work-stealing scheduler. Four threads are executed in parallel and executing the task from their local multi-queue. Whenever a scheduler runs out of local work, tasks can be retrieved from other threads. This is done round-robin.

Task Life-Cycle

Figure 3.5 shows the components used for the task life-cycle. The `TaskFactory` encapsulates all components required for the task creation. With the help of the factory, every task is capable of creating new tasks. These new tasks are distributed by the load balancer automatically. Since the load balancer may distribute the new task to any other thread, the queue must allow multiple producers.

3.2.2 Work-Sharing and Work-Stealing

As mentioned before, the load balancer is used to provide a mechanism for work-sharing [14]. Using the task factory and the load balancer together, every thread can create tasks for any other thread. However, for the most cases an optimal load balancing would be too expensive and thus a small imbalance will always remain. This imbalance may have two reasons: The first reason is, that tasks encompass different amount of work. This amount of work is not always known at compile time and can vary within the program. The second reason originates from the heterogeneity of the CPUs. Cores with different frequencies or a different set of features might compute tasks slower or faster.

This fact alone requires a dynamic scheduling at runtime. In the proposed task engine this is implemented using work-stealing [14] (see Figure 3.6). Whenever the local queue of a thread is empty and thus the thread runs out of work, the work-stealing procedure takes over. Round-robin all queues of all other threads are checked for available tasks. When a non-empty queue has been found, a task is retrieved and executed from this remote queue.

Nevertheless, work-stealing has a negative effect on data locality [1] namely cache-misses. But, compared to an idling thread, the runtime impact of a cache misses due to work-stealing can be neglected. Additionally, load-balancing is still possible and the user is encouraged to use both: load-balancing and work-stealing. The user is supposed to improve the load-balancing scheme. This will reduce the required work-stealing and thereby a trade-off between work-stealing and work-sharing can be reached.

3.3 Type-Driven Priority Scheduling

Latency-critical application often require a sophisticated scheduling to reach sufficient scaling. This scheduling must take special care of maintaining sufficient parallelism throughout the program. By scheduling the tasks in the right order some parallelization bottlenecks like starvation can be avoided.

Let's come back to the application of interest, the FMM, and its underlying data structure. Since it is a hierarchical method, it uses a tree-like data structure. Every vertex in the tree represents an object used for different tasks. This means, every level encompasses different numbers of vertices and thereby different amounts of parallelism. The bottleneck for the parallelization are the top levels in the tree encompassing only a few vertices. But as long as there are enough independent tasks on the lower tree levels, this bottleneck can be avoided by prioritizing the tasks on higher tree levels. Once the execution on a higher level does not expose enough independent parallelism, tasks from the lower levels will be executed instead. This makes a prioritization scheduler mandatory for the proposed task engine.

In contrast to most other tasks engines, the proposed framework uses typed tasks. Task performing operations upwards in the tree have a different type then tasks working on a specific tree level. This means, the type of the task represents a corresponding operation in the algorithm. For the scheduling this can be used to deduce the priority depending on the current operation and is called type-driven prioritization.

The core of the proposed scheduling is the type-driven priority queue. For the implementation of this queue two problems had to be solved. First, how to store diverse types of tasks in the same queue and second, how to prioritize itself.

3.3.1 Storing Tasks of Different Types

There are two classical solutions for storing tasks of different types in the same queue: The first uses function pointers and void pointer arguments and the second one uses virtual inheritance.

The first is a C-like approach. In this approach, function pointers are used for tasks. Those function pointers could have arbitrary function parameters. To store these function pointers in the same data structure, the signature of the function pointer needs to be equivalent. That is why only a single function parameter of the type void pointer would be allowed. Thus, the queue would store pairs consisting of a function pointer and a void pointer. For the execution of a task, the function would be executed by applying the void pointer as function argument. Inside the function body, the user must cast the void pointer to the desired type.

This approach has several drawbacks. It involves many type conversions for the argument and does not preserve type information. Additionally, those error-prone type conversions are done by the user. Giving up the type safety lowers the robustness of the code. Bugs from mixed parameters or faultily converted arguments arise at runtime and are very hard to debug.

The second one is a C++ approach. Instead of void pointers virtual inheritance

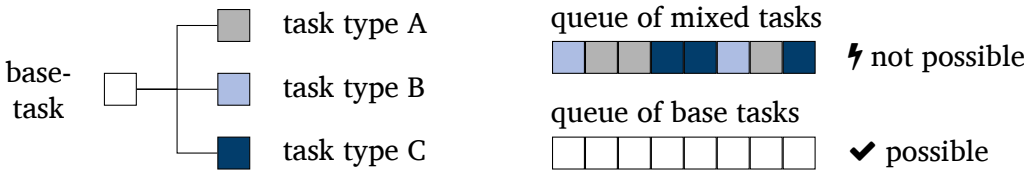


Figure 3.7: Storing tasks of different types in a single queue is not possible. Therefore virtual inheritance must be introduced to store the common base pointer.

would be used (see Figure 3.7). For this purpose a common base class is introduced. All tasks would need to derive from this base class. Each task requires an execute method using virtual inheritance. To store the tasks in a unique container, a pointer to the common base task would be used. As long as the execution of a task holds a significant amount of computation the costs of a virtual function call [34] can be neglected. Nevertheless, this approach still holds a drawback since virtual function calls cannot be inlined and thereby elude important optimizations.

In both solutions the type of the task is either completely lost via the void pointer or hidden by virtual inheritance. In the second approach it is possible to retrieve the type with dynamic casts, but this process comes with unacceptable overheads. For some parts of the type-driven priority queue the second approach will be used.

3.3.2 Priority Queues

For the scheduling of tasks along the critical path of any latency-critical algorithm a priority queue can be used. For the development of a new type-driven priority queue two classical solutions for the implementation of priority queues are discussed in the following.

Corresponding to [55, p. 351] priority queues can be subdivided into two groups. The first group describes bounded range priority queues using a discrete set of priorities, respectively a small set of priorities. The second group describes unbounded range priority queues with a very large set of priorities like all values of an integral type. A priority queue requires the following basic methods:

find-max Retrieve the element with the highest priority from the queue.

delete-max Remove the element with the highest priority from the queue.

insert Insert a new element into the queue.

These methods have different names in literature, but the semantic stays always the same.

Bounded Range Priority Queues

For applications requiring only a few priorities bounded range priority queues are sufficient. The restriction of available priorities leads to priority queues with lower complexities for the three main methods (see Table 3.1). A classical data structure for these queue is the bucket-based priority queue [33].

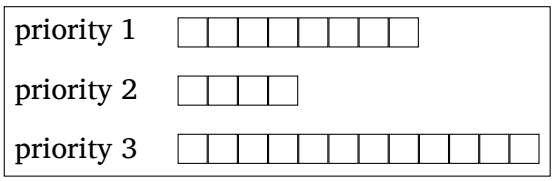


Figure 3.8: An example of a bucket-based priority queue consisting of three priorities.

Method	Complexity
find-max	$\mathcal{O}(1)$
delete-max	$\mathcal{O}(1)$
insert	$\mathcal{O}(1)$

Table 3.1: Bucket-based priority queue complexities of the three main methods.

A bucket priority queue consists of several ordered sub-queues (see Figure 3.8). The order of the sub-queues represents the priorities. The number of sub-queues and therefore the number of different priorities is denoted with k. Most often the priorities are represented as integral values:

$$0 \leq prio \leq (k-1)\,.$$

For inserting a new element, the correct sub-queue is retrieved by comparing the priorities of the new element and the sub-queues. This exhibits a maximum of k comparisons and since the number of sub-queues is fixed the complexity is $\mathcal{O}(1)$.

For the retrieval of the next element the sub-queues are iterated in the order of priorities. The first non-empty sub-queue is used and the next element is retrieved. This has the same complexity as the insert method of $\mathcal{O}(1)$. Strictly speaking, the prefactor is exactly the number of empty sub-queues before the first non-empty sub-queue.

This approach leads to a single-ended priority queue. However, a double-ended priority queue can be trivially implemented by iterating over the sub-queues in reverse order.

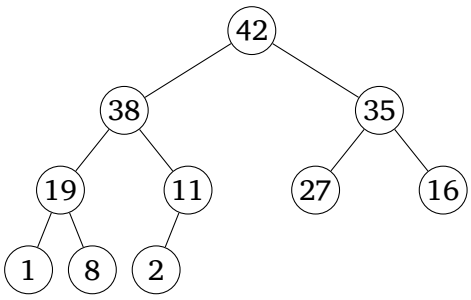
Most task engines provide priority queues with only a few priorities based on bucket priority queues (e.g. Argobots [102]).

Unbounded Range Priority Queues

Unbounded priority queues use a large set of priorities. For the implementation of those queues min and max heaps are often used. Since min and max heaps work similar, only max heaps will be discussed in the following.

Before describing max heaps, heap itself needs to be defined. The definition of a heap differs in the literature. The following definition is used in the remainder of this work and was adapted from [8].

Definition 3.5 *A **heap** holding n elements is a (binary) tree-based data structure*



(a) Max heap graph diagram

42	38	35	19	11	27	16	1	8	2

(b) Max heap memory layout

Figure 3.9: An example of a max heap graph diagram and the corresponding memory layout. The memory layout lists the tree levels from top downwards and the elements per level from left to right.

fulfilling the heap condition. The ***heap condition*** *requires that all leaves are at most on two levels.*

Heaps with n elements can be stored implicitly in an array of n elements [87]. For this purpose, the elements are listed top down, from left to right. Using this storage scheme, the root node can be called the first element and the rightmost element on the lowest level can be called the last element. Figure 3.9a shows an example of a max heap and the corresponding memory layout.

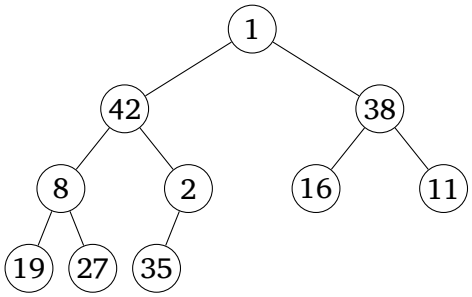
Definition 3.6 *A* ***max heap*** *is a heap with an additional ordering condition. For every node, the stored value is greater or equal to all of its descendants.*

Therefore, the maximum value can be found at the root node. This results in constant complexity for the find-max method (see Table 3.2). For inserting a new element, the element is inserted at the leftmost position on the lowest level. Afterwards it is compared to the value of the parent node. If the ordering condition is not fulfilled the elements are exchanged until the ordering is fulfilled or the root node is reached. This implies a maximum of $\log n$ comparisons and exchanges, hence the complexity for inserting is $\mathcal{O}(\log n)$. For deleting the maximum element the tree needs to be restructured to fulfill the heap condition again. For this, the last element is inserted at the root node. Afterwards it is compared to both child elements. If the ordering condition is not fulfilled it is exchanged with the smallest element of both. This will be continued until the ordering is fulfilled or a leaf node is reached. This exhibits logarithmic complexity. With this approach single-ended priority queues can be implemented.

For the implementation of double-ended priority queues a combination of min and max heaps is required. One combination is the min-max heap [8]. This combination of min and max heaps exposes the same complexity for the main methods (see Table 3.3). This means, it requires $\mathcal{O}(\log n)$ complexity for `insert`, `delete-min` and `delete-max` and $\mathcal{O}(1)$ complexity for `find-min` and `find-max`. To reach these complexities, an alternating order is used in the tree. The tree is

Method	Complexity
find-max	$\mathcal{O}(1)$
delete-max	$\mathcal{O}(\log n)$
insert	$\mathcal{O}(\log n)$

Table 3.2: Max heap-based single ended priority queue complexities of the three main methods.



(a) Min-Max heap graph diagram

1	42	38	8	2	16	11	19	27	35

(b) Min-Max heap memory layout

Figure 3.10: An example of a min-max heap and the corresponding memory layout.

distinguished into even and odd levels. For even levels the nodes are smaller or equal to their descendants (min heap) and for odd levels the descendants are greater or equal to their descendants (max heap). An example of a min-max heap can be found in Figure 3.10.

3.3.3 Type-Driven Priority Queue

Since the parallelization of the FMM requires only a few priorities a bounded range priority queue is sufficient for the development of the type-driven priority queue. The type-driven priority queue extends the bucket-based priority queue with the type-driven concept. Since the type-driven priority queue consists of several sub-queues it will be also called multi-queue in the following. Thereby the proposed implementation provides a faster insert method while providing equal performance for the find-max and delete-max methods compared to the bucket-based priority queue (see Table 3.4). Additionally, since the multi-queues relies on the type-system of C++ the application is more robust and queue-related

Method	Complexity
find-max	$\mathcal{O}(1)$
delete-max	$\mathcal{O}(\log n)$
insert	$\mathcal{O}(\log n)$

Table 3.3: Min-Max heap-based double ended priority queue complexities of the three main methods.

Method	Complexity
find-max	$\mathcal{O}(1)$
delete-max	$\mathcal{O}(1)$
insert	$\mathcal{O}(1)$

Table 3.4: Complexities of the type-driven priority queue for the three main methods.

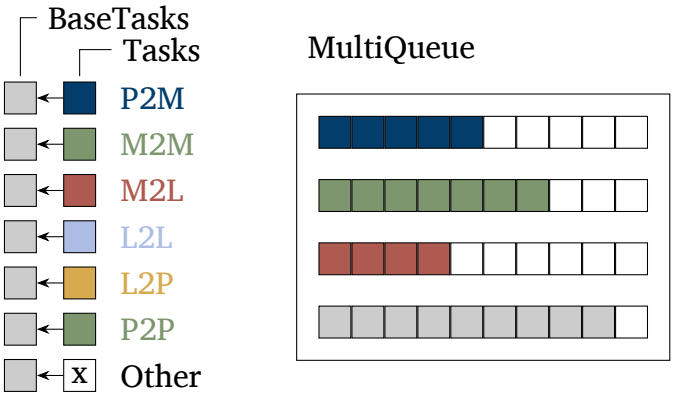


Figure 3.11: An example of a multi-queue. This multi-queue consists of three typed sub-queues for P2M, M2M and M2L. All other tasks are stored in the backfill queue using the common base pointer.

bugs can be quickly detected at compile-time.

Multi-Queue Design

An extension of the bucket-based priority queue used for the multi-queue is the representation of priorities with type-driven priorities. In the following the design of the proposed multi-queue is described. The multi-queue itself is only a facade for the underlying data structure. For the storage of tasks several sub-queues are used. Those sub-queues are implemented using the double-ended queue from the standard library (`std::deque`).

In the following the structure of the multi-queue is described in more detail. The multi-queue consists of several sub-queues as shown in Figure 3.11. All sub-queues store tasks with distinct types except the last sub-queue. Is is a special backfill queue for all tasks without priority. The priority of the tasks is given by the order of sub-queues like in bucket-based priority queues. The first sub-queue has the highest priority, whereas the last sub-queue has the lowest priority. The backfill sub-queues stores tasks of arbitrary type and uses virtual inheritance to accomplish that. The design is called herein a hybrid queue storage due to the mix of “typed” sub-queues and a sub-queue using a common base type also called backfill queue.

As previously discussed for the execution of the FMM only a few tasks are on the critical path and need to be prioritized. That is why only a few typed sub-queues

are required. To avoid overhead due to a large number of unnecessary sub-queues the backfill queue was introduced. Without the backfill queue a new sub-queue must be installed for each new task type. Some tasks are created only once in the simulation and do not require a high priority (e.g. data preparation or sorting).

One distinct feature of this multi-queue is the configurability at compile-time. At first this sounds like a disadvantage, since such a queue cannot adapt to changes at runtime. However most algorithms can define their data-flow and priorities upfront and like in the FMM these priorities will not change during runtime. For the configuration itself, an ordered list of priorities will be sufficient. If a task does not need a prioritization it can be neglected from this list. In such a case, the task will be stored in the backfill queue automatically. This requires all tasks to inherit from the base task pointer. To avoid the overhead of the virtual function call a special call syntax is used for the typed sub-queues. This will be explained in the implementation section.

Additionally, a certain task may require a LIFO or FIFO ordering on the corresponding sub-queue. Since all sub-queues are double-ended this can be achieved by inserting tasks on the corresponding end of the sub-queue.

3.3.4 Anatomy of the Type-Driven Priority Queue

In this section the challenges for the implementation of the multi-queue and the type-driven priority scheduling are discussed. Sticking to the differentiation of developer roles this is part of the library developers responsibility. Techniques shown in this section do not need to be understood by algorithm developers planning to use this library. The library tries to hide such complex implementation details, offering an easy-to-use interface.

The implementation of the type-driven priority scheduling and the multi-queue require several features of TMP. Especially, the “type-driven” resolution of priorities in a configurable manner is only possible using TMP. In the following, challenges arising from the given requirements are shown and their solution is discussed. An evolution of the multi-queue, from a simple proof of concept, towards a fully configurable and easy-to-use multi-queue is presented.

Composing Typed Sub-Queues

For a first proof of concept multi-queue the idea of bucket-based priority queues was used. The simplest implementation for this is the composition of different typed sub-queues to a single class. Figure 3.12 shows the class diagram of the simple multi-queue. In this example, FMM tasks are used, but it could be any set of typed tasks. The tasks required for the FMM are all considered for separate sub-queues. The find-max method iterates over those sub-queues in the order of priority until a non-empty sub-queue is found. The same applies for the delete-max method. For a multithreaded application it is required to find and delete the maximum value with an “atomic” method. That is why the two methods are fused to a single `find-delete-max` method. This method is used to retrieve and delete the next element in one method and can be implemented thread-safe using

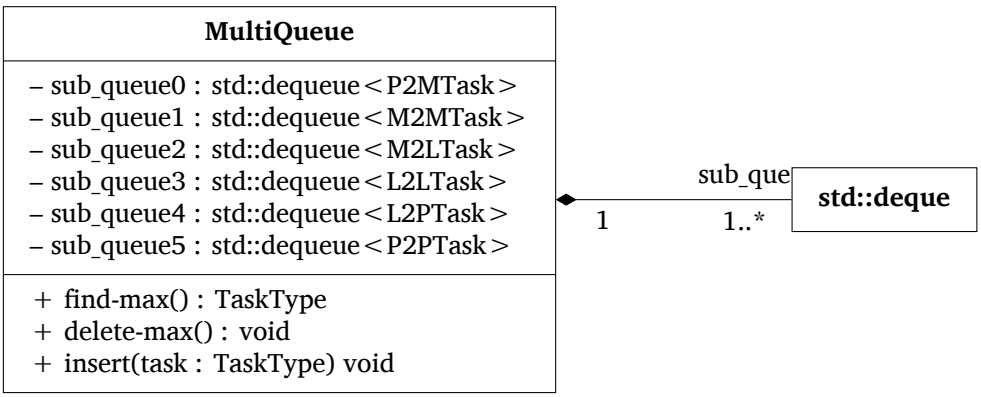


Figure 3.12: This is the UML class diagram of the simple MultiQueue consisting of fixed typed sub-queues only.

LISTING 3.4: SIMPLE MULTIQUEUE M2L INSERT METHOD

```
1 void Insert(Task<M2LProcessor<...>> * m2l_task) const
2 {
3     sub_queue2.push_back(m2l_task);
4 }
```

locking mechanism.

The insert method needs to be overloaded for the different task types to insert the new tasks into the correct sub-queue. Listing 3.4 shows the insert methods overload for M2L tasks. This simple prototype exhibits several problems for the use in a library. Due to the hardwired sub-queues together with the overloaded insert methods a configuration of the priorities is no trivial. For a new task type, the user would need to implement all required methods by hand. Changing the priority of a task would also require to change the underlying data-structure as well as the find-max method. Additionally, rarely executed tasks require separate sub-queues. This will increase the number of available but unnecessary sub-queues.

Those drawbacks can be eliminated and will be shown step by step in the following sections.

MAPPING TYPES TO VARIABLES

The most important feature for enabling the configurability of the multi-queue is the mapping of task types into corresponding sub-queues. A simple multi-queue uses several sub-queues. For every sub-queue a separate insert method is required. Also the find-max and delete-max methods must be adjusted if the sub-queues change. This is not only error-prone but also unacceptable from a usability standpoint of the algorithm developer. In contrast, the user interface of the multi-queue only requires a single list of tasks and the corresponding priority. Thus the coupling between defined priorities and the insert, find-max and delete-max methods need to be resolved.

For a more flexible implementation the C++11 class std::tuple is used. All

LISTING 3.5: ITERATING OVER THE ELEMENTS OF AN STD::TUPLE

```
1  template <typename sub_queue_t, int cur = 0>
2  static constexpr typename std::enable_if<cur != std::tuple_size<tuple_type>::value, int>::
       type
3  TuplePos()
4  {
5      return std::is_same<sub_queue_t, typename std::tuple_element<cur, tuple_type>::type>::
           value
6                 ? cur
7                 : TuplePos<sub_queue_t, cur + 1>();
8  }
9  template <typename sub_queue_t, int cur = 0>
10 static constexpr typename std::enable_if<cur == std::tuple_size<tuple_type>::value, int>::
       type
11 TuplePos()
12 {
13     return cur;
14 }
```

sub-queues are stored in a new sub-queue tuple and not in separate variables. The advantages of this approach is not obvious and needs further explanation. Instead of using numbered variable names like `sub_queue0, sub_queue1` the tuple abstracts the index and introduces a template parameter for this. Thus the `sub_queue0` becomes the element in the tuple at position zero and so on. Using this naming scheme, it becomes possible to iterate over elements of the tuple at compile-time. This makes a mapping between types and variables at compile-time possible in the first place.

Listing 3.5 shows the implementation of the iteration method. The method `TuplePos()` takes a template parameter called `sub_queue_t`. This is the type of the searched sub-queue. If the tuple holds a sub-queue of the same type, the position is returned, else the next position after the end of the tuple is returned. This works as follows: The method has two template parameters: `sub_queue_t` and `cur`. The non-type template parameter [114, pp. 45] `cur` has the default value of zero and reflects the current position of the tuple, starting with zero. The type of the sub-queue at the current position is compared to the template parameter `sub_queue_t` in line 5 using the type trait `std::is_same`. The type trait `std::is_same` compares two given types for equality and stores the result in the Boolean variable `value`. If the types are equal the current position is returned at line line 6. If the types are not equal the function is called recursively for the next position in the tuple (see line 7).

Furthermore, the selection of the correct sub-queue depends on the type of the sub-queue. The type must be known at compile-time. To enable the compile-time evaluation of the `TuplePos` method, it is marked as a constant expression (`constexpr`). Additionally, the TMP feature SFINAE is used (see Section 1.2) to test if the end of the tuple is reached. At line line 2 the current position (`cur`) needs to be different to the size of the tuple. This function is used in the case that the current position is in the bounds of the tuple. If the end of the tuple is reached, the second function starting at line 9 is used. This function returns the size of

LISTING 3.6: GENERIC INSERT METHOD OF THE MULTIQUEUE

```
1 template <typename task_type>
2 void push_back(task_type * task)
3 {
4     auto & queue = std::get<TupleType::TuplePos<task_type>()>(sub_queues_tuple);
5     queue.push_back(task);
6 }
```

the tuple and thereby stops the recursion. In this case, the sub-queue type could not be found in the tuple and does not exist. This will result in a compile-time error. From a maintainability point of view such compile-time errors are always preferred over runtime errors, since they show inconsistency at the earliest point possible.

With the help of the `TuplePos` method a generic insert method can be implemented shown in Listing 3.6. This method has a single parameter, a pointer to the task. The type of the task is reflected by a template parameter and automatically deduced. To retrieve the correct sub-queue corresponding to the task type, the `TuplePos` method can be used. The index returned by this method is used at line 4 to retrieve the correct tuple element. Since the tuple element is a standard double-ended queue, the task can be inserted as usual.

The enhanced multi-queue now offers the possibility of changing the priorities at a single line (the tuple definition). The insert method was decoupled and works generic for all definitions of priorities. A remaining disadvantage might occur for tasks without priority. In this implementation, every task type needs its own sub-queue, otherwise a compile-time error is emitted. A solution to this problem is given in the next section.

THE BACKFILL QUEUE

The backfill queue is used for tasks without priority. Those tasks can even be of different types and therefore the queue must use virtual inheritance to store such tasks. For the internal structure of the multi-queue this means, another sub-queue needs to be introduced. Tasks without priority are stored using the common base pointer in this backfill queue. In contrast to task in the typed sub-queue, tasks without priority are not stored with their distinct type in the backfill queue.

If the insert method resolves a valid typed sub-queue the task is inserted directly, otherwise if the insert method cannot resolve a valid typed sub-queue for the inserted task type, the task is inserted into the backfill queue automatically. Since the `TuplePos` method can be evaluated at compile-time, the same applies for the decision if a valid sub-queue exists.

The multi-queue is highly configurable and the user of the multi-queue can change the prioritization easily. Since the multi-queue must be capable of handling any task without prioritization, the virtual inheritance needs to be intrinsically available for all task types. This also means, that all tasks are executed using a virtual function call no matter whether they are from the typed sub-queue or the backfill queue. Even if the overhead from virtual function calls is negligible in

Listing 3.7: Test program for virtual function calls

```
1  #include <iostream>
2
3  struct Base {
4    virtual void foo() {
5      std::cout << "Base class called" << std::endl;
6    }
7  };
8
9  struct Derived : public Base {
10   virtual void foo() {
11     std::cout << "Derived class called" << std::endl;
12   }
13 };
14
15 int main() {
16   Derived * d = new Derived();
17
18   // enable one or the other to check vtable usage
19   d->foo();          // called  with vtable
20   d->Derived::foo(); // called directly without vtable
21 }
```

Listing 3.8: Difference in assembly of the v-table test program

```
1 <   movq  %rax, %rdi
2 <   call  _ZN7Derived3fooEv
3 ---
4 >   movq  (%rax), %rax
5 >   movq  (%rax), %rax
6 >   movq  -24(%rbp), %rdx
7 >   movq  %rdx, %rdi
8 >   call  *%rax
```

most cases, it should be avoided for tasks in typed sub-queues if possible.

Removing the Virtual Function Call

The overhead of a virtual function call is negligible as long as the work done inside the function is significantly larger than the costs of the call itself. Since, the task granularity is not known and the tasking framework should be as flexible as possible, overhead due to virtual inheritance should be avoided whenever possible. Additionally, the inlining capability of the compiler should not be impeded.

To avoid the virtual function call the complete call id can be specified and the virtual call is thereby suppressed. This means, the method remains virtual, but is called directly. Listing 3.7 shows a small test program combining both types of calls. The program was compiled using the GCC C++ compiler (version: 7.2.0) to x86_64 assembly in two different versions of the program. The first program uses the usual function call involving a v-table lookup (line 19). The second uses a direct call at line 20 which is supposed to be without a v-table lookup.

The resulting object dump is compared line by line and the difference is shown in Listing 3.8. The listing shows, that the version with the direct call removes the virtual function and calls foo directly, whereas the version with the virtual

LISTING 3.9: EXECUTETASK METHOD

```
1 void ExecuteTask(BaseTask * task) {
2   task->execute();
3 }
4
5 template <typename TaskType>
6 void ExecuteTask(TaskType * task) {
7   task->TaskType::execute();
8 }
```

function call resolves the call to a function pointer from the v-table.

EXECUTING TYPED TASKS WITHOUT A VIRTUAL FUNCTION CALL

To avoid unnecessary virtual function calls, the previous technique is used in the task engine. The multi-queue provides an execute task method. This method is used as a wrapper for executing a certain task from any sub-queue. This method can be specialized for the case, such that the task type is known.

Listing 3.9 shows the `ExecuteTask` method. The first method (line 1) is the implementation for task using a common base pointer. This method has a single function parameter of the type `BaseTask` pointer. This method calls the `execute` function without any further call path specification. The second method (line 5) uses the previously described call path specification. It uses a template parameter (`TaskType`) for the type of the task. This is a concrete task type and can be used to specify the call path. This removes the virtual function call.

The overload resolution of the actual function call of these two methods may need further explanation. It is not immediately clear which function is called for which object. The second method (line 5) is a function template. The function parameter could be a pointer to any type, including `BaseTask`. Additionally, as mentioned before all tasks inherit from `BaseTask`. This means, that all tasks could be implicitly converted towards the base class. However, during overload resolution at compile-time the non-template function is preferred over the function template [93]. That is why calling the method using a `BaseTask` pointer results in a call of the first function. Furthermore, overload resolution prefers candidates without implicit conversions. Thus the function template is called for concrete task types.

FIFO AND LIFO SUB-QUEUES

The sub-queues are implemented using the double-ended queue from the standard library. For this container type, elements can be inserted and extracted at both ends of the queue. Thereby FIFO or LIFO ordering only depends on the end used for extracting or inserting. For the implementation in the multi-queue the insert method is used to distinguish FIFO or LIFO ordering. The next task is always retrieved from the front.

First, let's take a look at the insert method (see Listing 3.10). It has one template parameter called `TaskType`. The only function parameter is a pointer to this `TaskType`. This task is inserted using the `push_back` method. This method

Listing 3.10: MultiQueue insert method primary template

```
1 template <typename TaskType>
2 void Insert(TaskType * task) const
3 {
4     queues_.push_back(task);
5 }
```

Listing 3.11: MultiQueue insert method template specialization

```
1 template <typename... ProcessorArgs>
2 void Insert(Task<M2LProcessor<ProcessorArgs...>> * m2l_task) const
3 {
4     queues_.push_front(m2l_task);
5 }
```

automatically selects the correct sub-queue and inserts the task at the end of this queue. Since the retrieve method always extracts the next task from the front, the default sub-queue has a FIFO ordering.

For selected task types, this can be adjusted to use LIFO ordering. Listing 3.11 shows a template-specialization of the same insert method. The function in this example is specialized for M2L tasks. Instead of using `push_back` for inserting the task, `push_front` is used. This changes the order of the sub-queue to LIFO.

Finding the next Task to Execute

For retrieving the next task from the queue, the sub-queues need to be iterated in the order of priority. After the first non-empty sub-queue has been found, a task can be returned.

As mentioned before, the sub-queues are all of different types and elements of an `std::tuple`. Iterating over a tuple using a for-loop is not possible. Elements of the tuple can not be retrieved by using runtime index variables like in a for-loop. The index of the element accessed must be specified using static, respectively compile-time index variables. With this in mind, mock-up version would look like nested if-then-else blocks, checking the sub-queues subsequently. These blocks nest, until the last sub-queue is checked for non-emptiness. Whenever a sub-queue is not empty, the task will be returned and the nesting will be exited. Unfortunately, this solution only works if the number of sub-queues stays constant and the order of priorities does not change.

For the multi-queue in the task engine the idea of nested if-then-else blocks was adapted using TMP. Instead of handwriting each nested block, the method recursively calls itself at compile-time and introduces another if-then-else block until the end of the tuple is reached. Listing 3.12 shows the implementation of the `ExecuteNextOrdered` method. The function has one template parameter `cur_pos`. This parameter reflects the current position checked in the tuple starting from the beginning (position zero). At line 4 the sub-queue at the current position is checked for emptiness. If the sub-queue comprises a task, this task will be executed. If the current sub-queue is empty, the current position is increased and

LISTING 3.12: EXECUTENEXTORDERED METHOD OF THE MULTIQUEUE

```
1  template <int cur_pos = 0>
2  typename std::enable_if<cur_pos != tuple_wrapper_type::tuple_size, void>::type
3  ExecuteNextOrdered() {
4    if (!tuple_wrapper.template GetQueue<cur_pos>().queue.empty())
5      ExecuteQue(tuple_wrapper.template GetQueue<cur_pos>());
6    else
7      ExecuteNextOrdered<cur_pos + 1>();
8  }
9  template <int cur_pos = 0>
10 typename std::enable_if<cur_pos == tuple_wrapper_type::tuple_size, void>::type
11 ExecuteNextOrdered() {
12   return;
13 }
```

the `ExecuteNextOrdered` method is called recursively. This will result in several nested function calls, but these calls are small and eventually will be inlined by the compiler. After inlining, the result is the same as handwritten nested if-then-else blocks. What happens if the order of priorities needs to be changed? The order of priorities can be modified by changing the order of sub-queues in the tuple. This will be adapted by the shown method automatically as the tuple is iterated from the beginning towards the end.

A remaining question can be: How will the recursion stop? This is similar to the `TuplePos` function in Listing 3.5. SFINAE is used to distinguish a current position representing an existing element of the tuple at line 2 and a position out of bound at line 10. The latter means, if the current position is equal to the size of the tuple, no sub-queue remains and the recursion stops.

PARALLEL ACCESS

Up until now, we only discussed a single multi-queue on a single thread using single consumer and single producer access scheme. For the task engine a multiple consumer and multiple producer queue is required. Thus the used synchronization strategy needs further considerations. Currently, the multi-queues can be accessed with different locks. Using locks instead of lock-free queues is less error-prone and not performance critical for the considered use case. Nevertheless, since the underlying data structure can be exchanged it is possible to introduce lock-free queue implementations from other third-party libraries like [13, 32].

3.3.5 TYPE-DRIVEN PRIORITY SCHEDULER

For the scheduling the type-driven priority scheduler uses one multi-queue per thread. This multi-queue is assigned to a thread and resembles the thread's local multi-queue. Nevertheless, this local multi-queue is not fully private since it can be accessed by any other thread via the load balancer. Thus it is capable of work-sharing and work-stealing. The scheduling itself is done by the multi-queue. This means, the scheduler can simply extract the next task for the execution from the multi-queue as follows (see Figure 3.13). This scheduling algorithm is used for all threads. If the main multi-queue of the thread is not empty, the scheduler uses

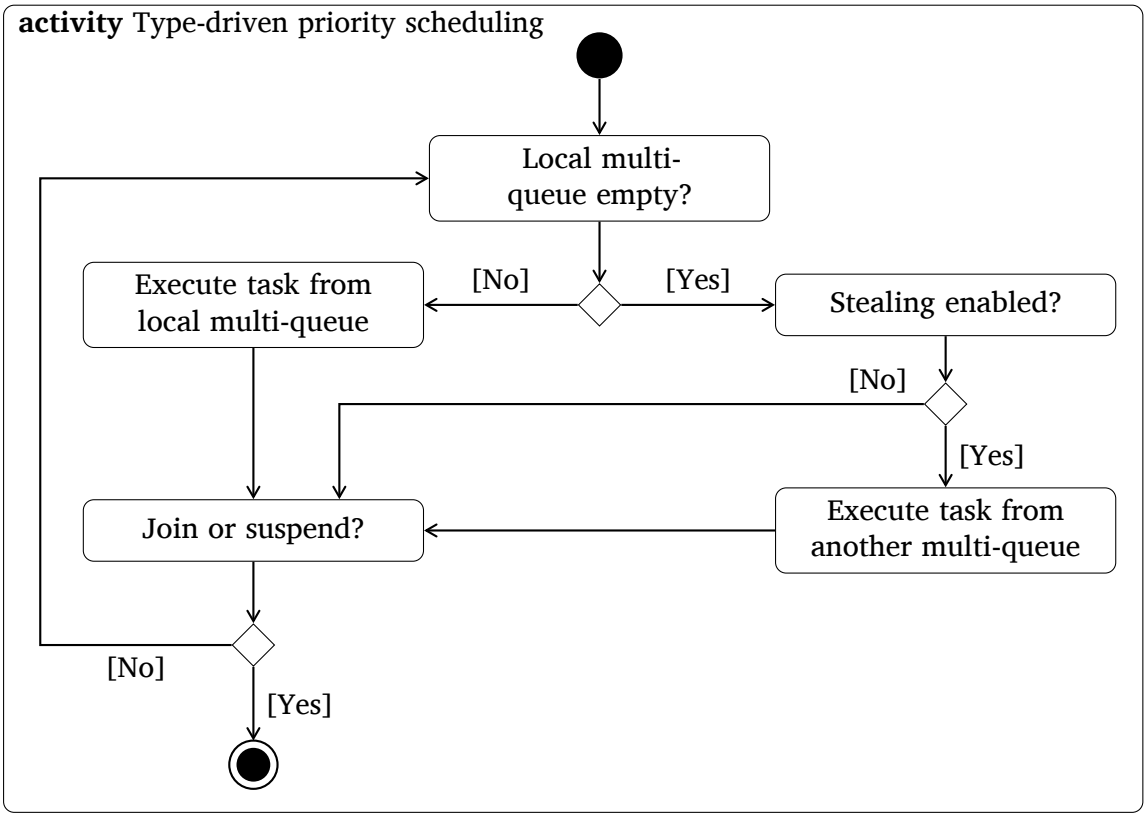


Figure 3.13: The UML activity diagram of the type-driven priority scheduler. It shows the flow of the scheduling activity. The polling-for-work loop is executed by each thread separately. The prioritization of the task is done internally by the multi-queue and therefore is not part of the scheduling itself. Whenever the flag `join` is set, the threads will join or suspend.

LISTING 3.13: EXAMPLES OF THE MULTIQUEUE DEFINITION

```
1 // example A, three priorities
2 using MultiQueueType = MultiQueue<TaskA, TaskB, TaskC>;
3 // example B, three priorities
4 using MultiQueueType = MultiQueue<TaskC, TaskA, TaskB>;
5 // example C, one priority
6 using MultiQueueType = MultiQueue<TaskA>;
```

the find-max and delete-max method to retrieve and delete the next task from the queue. Afterwards, the task will be executed. If the main multi-queue of the current thread is empty and work-stealing is enabled, the work-stealing is started. In this case, the next multi-queue is checked in ascending order on the thread id.

Finally, the scheduler checks the join flag. If the flag is set to true, the threads join or suspend. Otherwise the scheduling starts again by selecting a new task.

CONFIGURING THE TYPE-DRIVEN PRIORITY SCHEDULER

In this section an overview of the interface of the multi-queue and the type-driven priority scheduler is given. Corresponding to the categorization of developers for HPC this is the high level algorithm developers point of view. The usage of the interface requires to apply template parameters, but does not require the understanding of the inner working of TMP itself. The goal of the described high level interface is usability without compromising performance. This was achieved by a high level of abstraction of the internals of the multi-queue. The underlying implementation is completely hidden from the user.

Listing 3.13 shows three different example definitions of the multi-queue. Such a definition is the sole configuration point for prioritization in the type-driven priority scheduling. The first definition will prioritize the task in the order `TaskA`, `TaskB` and `TaskC`. If the user wants to change this order, only a change of the keywords is required. This is shown with the second multi-queue definition. In contrast to the first it will prioritize `TaskC` with the highest priority. Additionally, it is possible to omit tasks completely. This is done in the third example. In this example, only `TaskA` will be prioritized. All other tasks are internally stored in the backfill queue.

Thus the list of task types in the definition of the multi-queue defines the priority of tasks as follows:

1. Only tasks listed in this definition get prioritized.
2. The order of definition defines the priorities.
3. Tasks not listed will be scheduled without priority.

The multi-queue itself features two insert methods: `push_front` and `push_back`. The only parameter of these methods is a pointer to a task of any type. These methods determine the correct sub-queue for inserting either at the end or the beginning of the sub-queue. The internal selection of the correct sub-queue is

done at compile-time without further interaction by the user. If the provided task does not have a typed sub-queue, the backfill sub-queue is selected as fallback at compile-time. Therefore the call to these two insert methods directly chooses the correct sub-queue.

3.4 Static Data-Flow Dispatcher

In the previous Section 3.1.3 the data-flow graph was presented. This data-flow graph can be utilized to model and describe algorithmic dependencies. It consists of edges representing the data and vertices representing the algorithmic operations performed on that data. For the data-driven execution of an algorithm, the corresponding data-flow graph needs to be mapped into the task engine.

This can be done with the static data-flow dispatcher proposed in this section (see Figure 3.14). The static data-flow dispatcher works as follow: Data-events like the computation of a multipole can be triggered at the static data-flow dispatcher by calling the dispatch method. The dispatcher will forward the triggered event to all event handlers registered for this event. The following sets are used for the definition of the static data-flow dispatcher:

- A set of *data-events* (e.g. a multipole) mapping the edges of the data-flow graph.
- A set of callable *event handlers* (e.g. M2L task) mapping the vertices of the data-flow graph.

To register an event handler at the dispatcher, the handler and the event need to be combined in an event listener.

Definition 3.7 *An **Event Listener** is the combination of a data-event and an arbitrary number of event handlers.*

Therefore the data-flow dispatcher itself is a collection of event listeners.

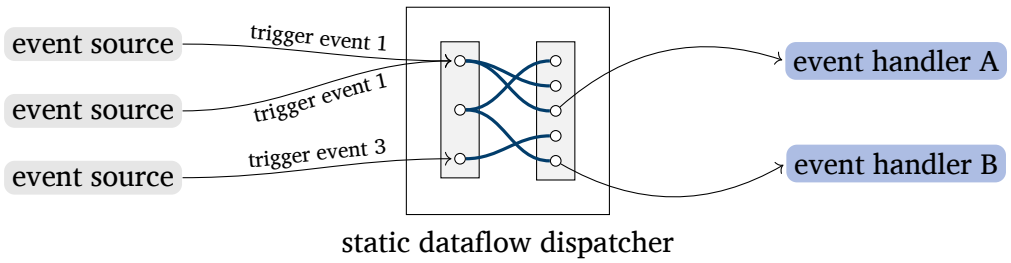


Figure 3.14: The schematic view of the static data-flow dispatcher. The left hand side of the switch board represents the data-events and the right hand side represents the event handlers. The wiring represent the registered event listeners. When an event is triggered by an event source, all corresponding event handlers are executed. The static part of the static data-flow dispatcher is the blue wiring itself. This represents the dispatch call, which is completely resolved at compile-time.

LISTING 3.14: DATA-EVENTS AND EVENT HANDLERS

```
enum DataEventSet {
  DataEvent1,
  DataEvent2,
  DataEvent3,
};

enum EventHandlerSet {
  TaskA,
  TaskB,
  TaskC,
  TaskD,
};
```

LISTING 3.15: CALLABLE HELPER CLASS FOR EVENT HANDLERS

```
template <EventHandlerSet EventHandler>
struct CallableImpl {};

template <>
struct CallableImpl<TaskA> {
  static void Call(size_t id, TaskFactory task_factory) {
    // Create TaskA
  }
};
```

Definition 3.8 *A **Data-Flow Dispatcher** is a set of event listeners and a dispatch method.*

3.4.1 ANATOMY OF THE STATIC DATA-FLOW DISPATCHER

In order to understand the components of the high level interface of the static data-flow dispatcher, some details of the implementation have to be discussed first. Therefore, the library developer's view will be explained first and the view of the algorithm developer is shown afterwards.

The data-flow dispatcher must be configurable at compile-time. To enable compile-time configuration, the following C++ type definitions model the static data-flow dispatcher.

The definition of the data-flow dispatcher requires two sets:

- the first set encompasses the data-events and
- the second set encompasses the callable event handlers.

These sets are defined by using enumerations. Listing 3.14 shows an abstract example using three data-events and four event-handlers. These definitions are the basis for creating the data-flow dispatcher.

As mentioned in the definition, the event handlers need to be executable. This functionality is done with a helper class called `CallableImpl`. Listing 3.15 shows an example of this helper class. The class has a template parameter from the set of event handlers. To provide the callable feature of an event handler, the class

Listing 3.16: Definition of the EventListener Type

```
1 template <DataEventSet RegisteredEvent, EventHandlerSet... EventHandlers>
2 class EventListener {
3   // ...
4 };
5
6 // Example
7 using EventListener1 = EventListener<DataEvent2, TaskC>;
8 using EventListener2 = EventListener<DataEvent1, TaskA, TaskB>;
```

Listing 3.17: EventListenerContainer for collecting EventListeners

```
1 template <typename... EventListeners>
2 class EventListenerContainer {
3   // ...
4 }
```

needs to be specialized and a static `Call` method needs to be provided. This is done at line 4 for the event handler `TaskA`. With this class, the event handlers become callable.

Until now, it is only possible to define a set of data-events and a set of callable event handlers. For the definition of a data-flow dispatcher the composition of data-events and event handlers into the event listeners is required. Listing 3.16 shows the definition of the event listener type. The class `EventListener` has two template parameters. The first is the registered event and the second is an arbitrary number of event handlers. Listing 3.16 also shows two examples of concrete event listeners. The first uses the data-event `DataEvent2` and the event handler `TaskC`. The second example combines the data-event `DataEvent1` and the event handler `TaskA` and `TaskB`.

The last feature missing for the static data-flow dispatcher is the possibility to collect event listeners. This is done using the event listener container shown in Listing 3.17. The `EventListenerContainer` uses a variadic template parameter for arbitrary many event listeners.

The Dispatch Flow

Using the data structures shown before the structure of the static data-flow dispatcher can be defined. The only thing missing is the actual dispatch functionality. This will be introduced now.

The desired dispatch workflow can be seen in Figure 3.15. The `EventListenerContainer` and the `EventListener` are extended with a static dispatch method. The top level dispatch method is the method called by the algorithm developer and implemented in the `EventListenerContainer`. This method will iterate over all registered `EventListeners` and call their dispatch method. The event listeners will filter if the triggered event is equal to the registered event. If the events are the same, the event listener will call the callable helper class for all event handlers. If the events are not the same, the event listener will immediately return. Since the data-flow structure is defined as template parameters, the iteration as well as

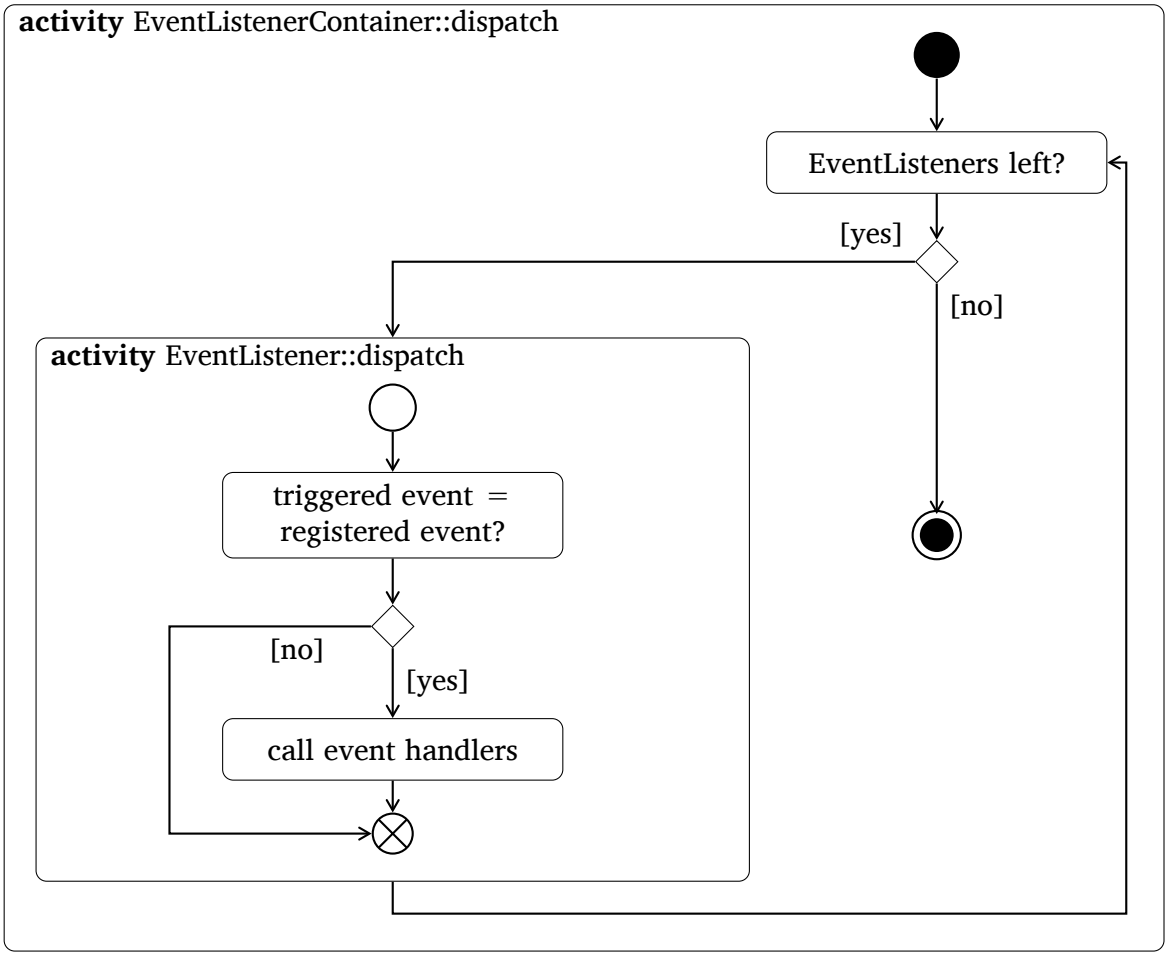


Figure 3.15: The flow of the dispatch method of the `EventListenerContainer`. The `EventListenerContainer` iterates over all available `EventListener` and calls their dispatch methods. The `EventListener` calls the event handlers if the events match.

Listing 3.18: EventListener template

```
1  template <DataEventSet triggered_event, typename... Args>
2  static typename std::enable_if<triggered_event == registered_event, void>::type
3  dispatch(Args &&... args) {
4    // Events are the same.
5    // Call all event handlers
6  }
7
8  template <DataEventSet triggered_event, typename... Args>
9  static typename std::enable_if<triggered_event != registered_event, void>::type
10 dispatch(Args &&...) {
11   // Events are different
12   // NO-OP
13 }
```

the filtering must be done using compile-time programming.

Filtering Events

The dispatch process needs to filter the event listeners at some point. This is done by the event listeners itself. Each event listener will distinguish between two cases. Either the triggered event and the registered event are the same or not. If this filtering can be done at compile-time, the resolving of the dispatch method can be done at compile-time as well.

Listing 3.18 shows the dispatch method of the event listener. The interface of this method and the template parameters are the same as for the dispatch method of the `EventListenerContainer`. The event listener uses SFINAE to filter the events (at line 2 and line 9). The first implementation of the dispatch method is valid for the case, that the triggered event is the same as the registered event. In this cases, the dispatch method will call all event handlers belonging to this event listener. The second method implementation is valid for the case, that the events are different. This means, the event listener is not registered for the triggered event. Thus it results in an empty method. These empty methods will be removed by the compiler.

3.4.2 Static Data-Flow Dispatcher User Interface

How can the previously defined types and methods be used by the algorithm developer to configure the static data-flow dispatcher? The user interface sticks to the formal definition of the static data-flow dispatcher. The user starts by defining the set of data events and event handlers like in Listing 3.14. After these sets are defined, the callable methods of the event handlers need to be implemented as shown in Listing 3.15.

After these definitions of the data events and event handlers, the configuration of the static data-flow dispatcher can start. Listing 3.19 shows an example of a static data-flow dispatcher. This dispatcher encompasses three event listeners. The event listeners represent the data-flow graph excerpts shown in Figure 3.16.

Additionally, the call of the dispatch method for the `DataEvent2` is shown. This dispatch call is completely resolved at compile time. After the substitution of

LISTING 3.19: AN EXAMPLE OF A STATIC DATA-FLOW DISPATCHER

```
using StaticDataFlowDispatcher =
    EventListenerContainer<EventListener<DataEvent1, TaskA>,
                           EventListener<DataEvent2, TaskC, TaskB>,
                           EventListener<DataEvent3, TaskA, TaskB, TaskD>>;

// Computation ...

// Example dispatch of DataEvent2
StaticDataFlowDispatcher::dispatch<DataEvent2>(var);

// After template substitution by the compiler and inlining this will result in:
CallableImpl<TaskC>::Call(var);
CallableImpl<TaskB>::Call(var);
```

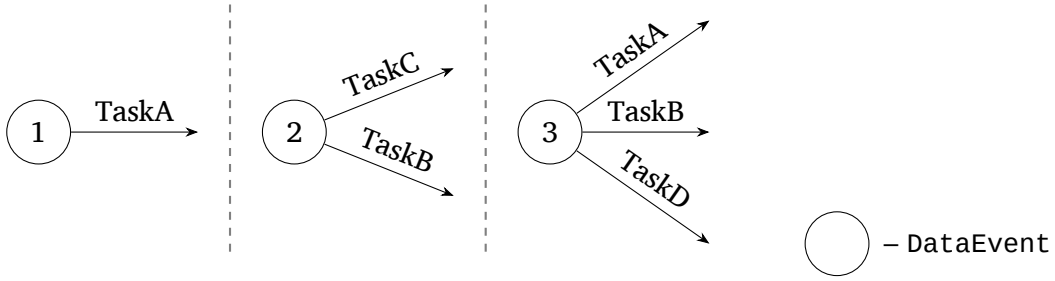


Figure 3.16: Excerpts from the data-flow graph corresponding to the example of the static data-flow dispatcher in Listing 3.19.

template parameters and the inlining the resulting calls are shown at the end of the listing.

THE STATIC PART OF THE DISPATCHER

What is static part in the static data-flow dispatcher? The static refers to the compile-time resolution of the dispatch method (see Figure 3.14). The "wiring" between the event and the corresponding event handlers is done at compile-time. However, this does not restrict the user to call the dispatch method at runtime. After the compilation the resulting code will not contain the dispatch call anymore, but instead only the direct call to the corresponding event handlers. This means, at runtime no expensive resolution of the dispatch method is required anymore. This lowers the overhead introduced by the dispatcher to a minimum.

3.4.3 DEPENDENCY COUNTER

As the name implies, the static data-flow dispatcher dispatches data-events. These data-events can either be triggered or not. But some applications might require to trigger an event several times, before the actual dispatch can be done. In this section the static data-flow dispatcher is extended with dependency counters fulfilling this need.

Figure 3.17 shows this extension using an interposed counter. Figure 3.17 (a) represents the direct triggering of a data-event and Figure 3.17 (b) shows the delayed event triggering through a dependency counter. As presented in this

(a) Without Counter (b) With Counter

Figure 3.17: Without a dependency counter, the data event can be triggered directly. By introducing a dependency counter, the trigger is delayed with respect to the counter.

LISTING 3.20: DEPENDENCYCOUNTER WRAPPER CLASS

```
1 template <DataEventSet event>
2 struct DependencyCounter {
3
4   //...
5
6   std::atomic<std::size_t> counter_;
7 };
```

example, all data-events will be extended by a dependency counter. When the data-flow dispatcher dispatches an event, the dependency counter of this event will be decremented and only if the counter becomes zero, the actual event is dispatched.

The data-flow dispatcher should be highly configurable. Therefore it is required, that the dependency counters are generated automatically from the configuration of the data-flow dispatcher. Again, this is done with the help of TMP. For every data-event used in the configuration of the data-flow dispatcher a separate counter will be created. As an example the automatically generated dependency counters for the static data-flow dispatcher from Listing 3.19 would encompass counters for `DataEvent1`, `DataEvent3` and `DataEvent2`.

The counters itself are encapsulated in a wrapper class called `DependencyCounter`. This encapsulation is required to allow user-defined default values. The implementation of the `DependencyCounter` class is shown in Listing 3.20. This class uses atomics from the standard library [68, pp. 1012] by default to allow thread safe concurrent counting.

Since the counters are generated at compile-time it becomes complicated for the user to define their default values. But for the most algorithms, it is required to set algorithm-specific initial values for different data-events. This can be done with the helper class shown in Listing 3.21. The value provided in this class will be used as an initial value for the specific data-event dependency counter. To set a default value for a data-event, this class must be specialized for this data-event.

LISTING 3.21: DEFINITION OF DEFAULT VALUES FOR DEPENDENCY COUNTERS

```
template <DataEventSet event>
struct DependencyCounterDefaults {
  // Without specialization, all counter are set to 1
  const static size_t value = 1;
};

template <>
struct DependencyCounterDefaults<DataEvent1> {
  // Counter for DataEvent1 is set to 100
  const static size_t value = 100;
};
```

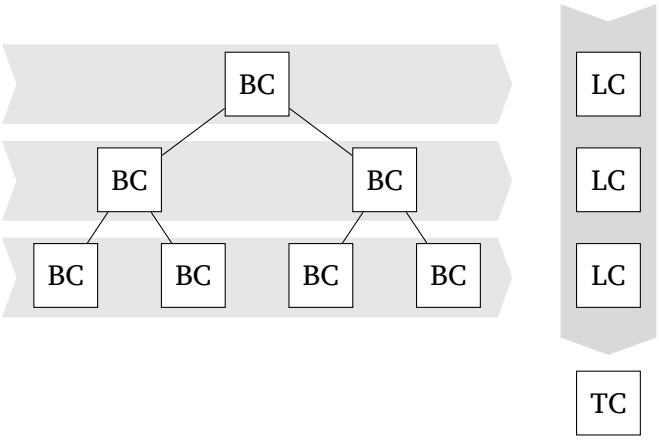


Figure 3.18: If a box counter (BC) becomes zero, the level counter (LC) will be decremented automatically. If the level counter becomes zero, additionally the tree counter (TC) will be decremented.

HIERARCHICAL COUNTERS AND DATA EVENTS

Not all applications have such simple dependencies. For hierarchical methods like tree codes or the FMM it makes sense to extend the concept of data-events and counters with respect to the surrounding data structure hierarchy. For the FMM this would mean, a data-event can be triggered for a vertex in the tree, a complete level in the tree or for the entire tree. For this purpose, a set of hierarchy levels is defined (e.g. vertex, level, tree) and the event listener is extended to support these hierarchy levels.

Definition 3.9 *A **Hierarchical Event Listener** is the combination of an event listener and a hierarchy level.*

A hierarchical event listener is only registered for the data-event on the corresponding hierarchy level. An example of this idea for a tree-based structure is shown in Figure 3.18. This tree has three levels and seven vertices. The set of hierarchy levels encompasses vertex, level and tree. For each vertex, level and tree a data-event is defined and a dependency counter will be created. Only the data-events on the lowest level in the hierarchy can be triggered by the user. In

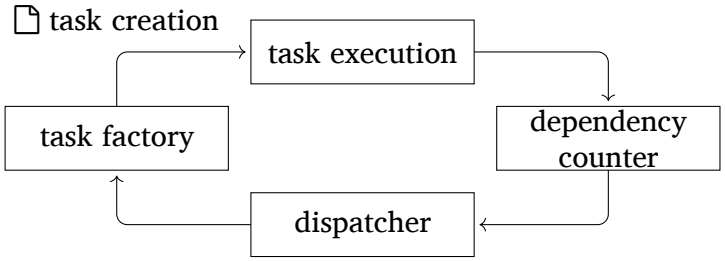


Figure 3.19: The flow of the dependency manager. After a task was executed, the corresponding dependency counters are decremented. If a dependency counter becomes zero, the dispatcher will be invoked. The dispatcher will create a new task using the task factory if necessary.

this example these are data-events on the vertex level. All other data-events (level data-events and tree data-events) are derived and will be triggered internally.

The hierarchical counter work-flow is as follows: Whenever a vertex counter becomes zero, a data-event for the vertex is triggered. Additionally, the counter for the next hierarchy level is notified. For the tree example this would be the level counter. In this way, the data-event propagates upwards in the hierarchy. When all data-events of a level have been triggered, the data-event will be triggered for the level itself. This will also notify the counter for the entire tree. When all levels triggered the data-event for the level, the data-event for the entire tree will be triggered automatically.

3.4.4 Event Handlers

As shown Listing 3.15, an event handler could execute an arbitrary function. This might suggest the user has to implement algorithmic computation inside the event handler. This is not true. Algorithmic computation should only be implemented in tasks itself. Tasks can be load-balanced and dynamically scheduled, event handlers cannot.

An event handler should only create new tasks and enqueue these tasks. The desired workflow is shown in Figure 3.19. After the execution of a task, the dependencies are resolved by decrementing the dependency counter. If the counter becomes zero, a data-event is triggered at the static data-flow dispatcher. If an event listener is registered for the triggered data-event, the event handlers will be executed. The event handler uses the task factory to create a new task and hands it over to the load balancer. The load balancer then will decide where to schedule the task and will insert the task into the corresponding multi-queue.

3.5 Concurrent Data Access

Concurrent data access is one of the most important challenges in parallel programming. If multiple threads want to manipulate the same piece of data, synchronization must take place. In this section several topics related to the data access in a task-parallel shared memory program are discussed.

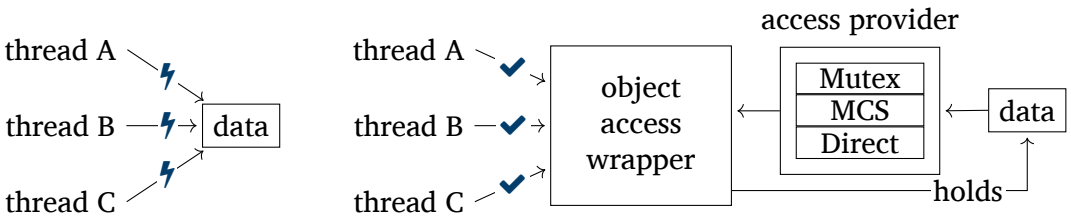


Figure 3.20: Concurrent access to data using the object wrapper. The left figure shows the direct access to the data potentially introducing data-races. Using the object wrapper, the access can be acquired using different strategies. This ensures thread-safe data access.

The first part is about the library's capabilities of wrapping the data-access to handle the actual access in an abstract way. Afterwards, synchronization and cache coherence will be discussed. An additional section will put the focus on a special synchronization strategy, namely locks. Finally, the awareness of non-uniform memory access in the algorithm and the task engine is discussed.

3.5.1 Encapsulation of Data Accesses

To avoid pitfalls related to data-races the task engine offers capabilities for data access in a thread safe manner. This is achieved with the so called `ObjectWrapper`. Instead of directly accessing data concurrently the user must acquire the access using the object wrapper.

For this purpose the object wrapper uses internal object access strategies like lock-based strategies. These strategies are exchangeable at compile-time. Additionally, it is possible to provide user-defined strategies.

Implementation

For the object access it is important to enable exchangeable access strategies. With the object access wrapper it is possible to use other locks or lock-free and wait-free algorithms. The task engine itself offers a mutex lock and an MCS lock access strategy. The implementation should be configurable by template parameters defining the corresponding data type and the desired access strategy.

Listing 3.22 shows parts of the object wrapper implementation. The object wrapper has two template parameters: the `ObjectType` and the `ObjectAccessStrategy`. The type of the object wrapped by this class is reflected by the `ObjectType`. The object access provider is determined by the template parameter `ObjectAccessStrategy`. The object access provider is used to acquire the access to the object using different strategies.

The object wrapper stores a reference to the original object and creates an access provider for it. The `AcquireAccess` method grants access to the object using the defined access strategy. Technically, it calls the acquire method of the access provider. The access strategy needs to be implemented as a specialization of the object access provider. This is done for the mutex lock and the MCS lock. Other strategies would require a specialization written by the user.

LISTING 3.22: IMPLEMENTATION OF THE OBJECTWRAPPER

```
template <typename ObjectType, typename ObjectAccessStrategy>
struct ObjectWrapper {
  using object_access_provider_type =
      ObjectAccessProvider<ObjectType, ObjectAccessStrategy>;
  using object_access_type =
      typename object_access_provider_type::object_access_type;

  const object_access_type acquire_access(const ThreadIdentifier & thread_id) {
    return oap_.acquire(thread_id.ID(), obj_);
  }

private:
  ObjectType & obj_;
  object_access_provider_type oap_;
};
```

LISTING 3.23: MUTEX LOCK SPECIALIZATION OF OBJECT ACCESS STRATEGY

```
template <typename ObjectType>
struct ObjectAccessProvider<ObjectType, MutexLockStrategy> {
  using oa_t = ObjectAccess<object_type, MutexLockStrategy>;

  const oa_t acquire(const user_id_type, ObjectType & obj) {
    mutex_.lock();
    return oa_t(obj, *this);
  }

  void release() {
    mutex_.unlock();
  }

private:
  std::mutex mutex_;
};
```

Listing 3.23 shows a simple example of an object access provider using a mutex lock for the synchronization. Whenever a thread wants to acquire access to this object the acquire method will be called. In this example the mutex will be locked and the `ObjectAccess` is returned. The object access itself is mainly used to release the lock in the destructor and is explained next.

Listing 3.24 shows the implementation of `ObjectAccess` class. It is similar to a pair with a reference to the acquired object and the used access provider. In the destructor of the object access, the release method of the access provider is called. This leads to an exception safe implementation of the acquired resources. Whenever another exception occurs, the destructor will be called and the lock will be released.

3.5.2 CACHE COHERENCY

CPUs use private caches and therefore need to guarantee the consistency of these caches. Whenever data resides in one cache and is manipulated in another cache, the consistency of the data needs to be maintained at all times. This is done utilizing cache coherence protocols. Usually, those protocols work on junks of

LISTING 3.24: OBJECTACCESS USED FOR MUTEX LOCK STRATEGY

```
template <typename ObjectType>
class ObjectAccess<ObjectType, MutexLockStrategy> {
public:
  ObjectAccess(ObjectType & o,
               ObjectAccessProvider<ObjectType, MutexLockStrategy> & oap)
    : obj_(o), oap(oap){};

  ~ObjectAccess() {
    oap.release();
  }

  ObjectType & Obj() const {
    return obj_;
  }

private:
  ObjectType & obj_;
  ObjectAccessProvider<ObjectType, MutexLockStrategy> & oap;
};
```

data (cache lines) instead of single data elements. For the Intel Xeon SP processors such a cache line contains 64 bytes.

A well known cache coherence protocol is the MESI protocol, also known as the Illinois protocol [94]. Figure 3.21 shows the algorithmic flow of the MESI protocol, adopted from [94, p. 350]. This protocol uses the following states for every cache line:

Exclusive-Modified (M) This cache line is owned exclusive and has been modified, hence it is not consistent with main memory. A write back to the memory is required.

Exclusive-Unmodified (E) This cache line is owned exclusively and is unmodified. It is consistent to the data in main memory.

Shared-Unmodified (S) This cache line is shared and hence not available exclusively by one core. It is unmodified and hence consistent with the main memory.

Invalid (I) This cache line is invalid.

Even if current CPUs might not use the original MESI protocol, cache coherence is required for concurrent data access. This means, writes require an invalidation of all other caches if the data was in a shared state.

This means, besides the usual cache misses triggered by the application cache invalidations due to the cache coherence protocol will occur. Especially for global variables involving many read and write operations from concurrent threads this may cause a performance bottleneck. While read access only requires a shared cache line, write operations always need exclusive access before writing. A typical scenario for global variables requiring write access are global counters and locks based on read-modify-write operations.

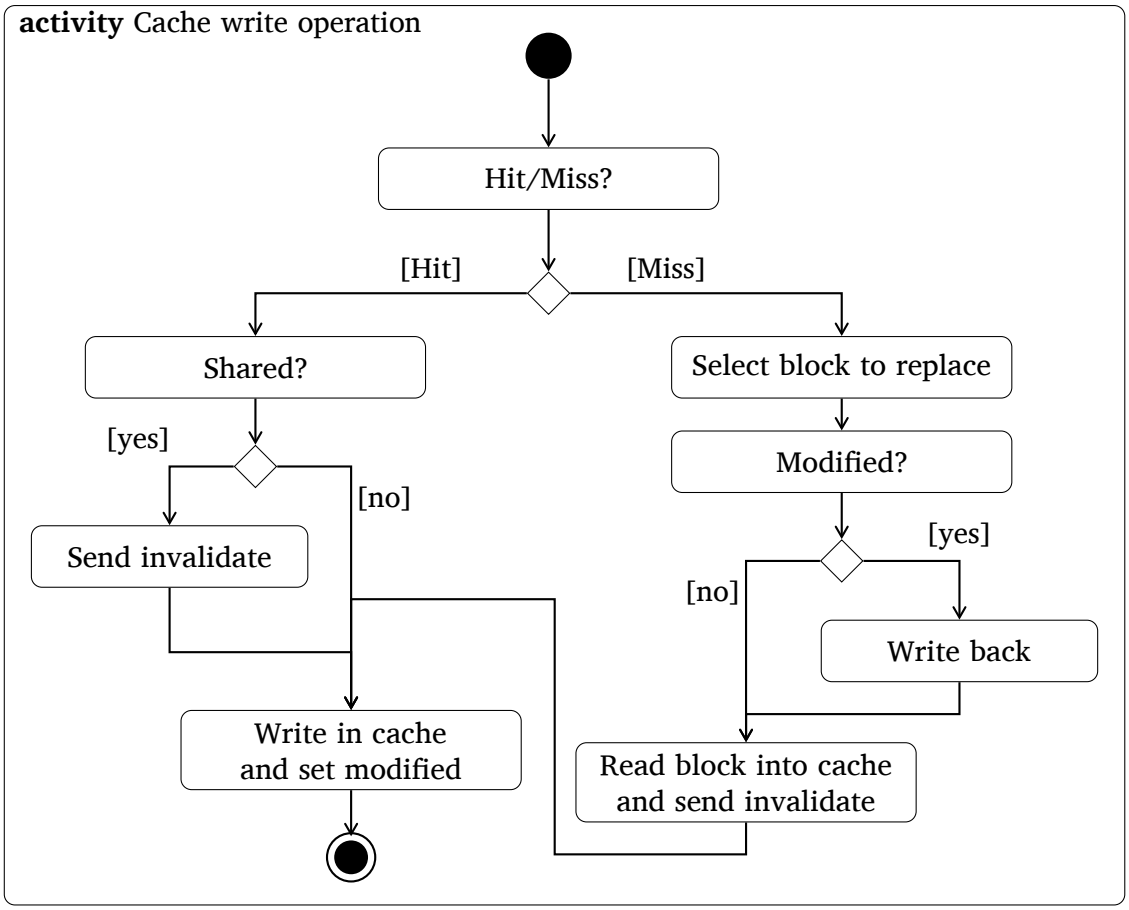


Figure 3.21: The activity diagram for maintaining the cache coherence for a write operation using the MESI protocol adapted from [94, p. 350]

For compute nodes with multiple sockets this problem gets even worse. Here the cache coherence protocol cannot use the last level cache but must use a slower bus connection between the sockets to restore coherency. Therefore, cache invalidations crossing NUMA borders become even more expensive. This means, for locks cache coherence friendly concepts are required. Such schemes are discussed in the following sections.

3.5.3 Locks

Locks are a basic tool for thread safe data-access in concurrent programs. Locks ensure, that the execution of a critical section by one thread does not overlap with the execution of the same critical section by another thread. Referring to [55, p. 24] this is called the *mutual exclusion* property.

Many lock implementations use busy waiting for the exclusion of threads. This may cause unnecessary contention either on the last level cache or the bus between different NUMA nodes. In the following some locks are presented and discussed with respect to performance.

Mutex Locks

A mutex lock is the default lock in the pthreads library [89] and available in the threads support library of C++11. The initial state of a mutex lock is unlocked.

A thread acquiring the lock calls the lock method and will succeed, if no other thread has acquired the lock. If the lock is already locked, the mutex uses a so called wait queue. The acquiring thread will be added to the wait queue and suspended until the lock is released. For short lock contentions this suspension mechanism introduces overhead. Additionally, this lock relies heavily on system calls for the synchronization. This can be avoided by using the fast user level locking API of Linux (Futex) [44]. The mutex locks in the pthreads library as well as the mutex in the standard library use futexes. Nevertheless, in the case of contention, expensive system calls for the thread suspension are required.

Spin Locks

Spinlocks are basic locks for shared memory programming. A spinlock encompasses a single Boolean compare-and-swap (CAS)-register for its synchronization. In C++ a Boolean CAS-register is implemented as an atomic flag, which is a Boolean variable providing a CAS method.

The Boolean variable shows whether the lock is in a locked state or not. The initial state is unlocked and thus the Boolean flag is set to false. Whenever a thread wants to acquire the lock, it repeatedly executes CAS on the atomic flag until it succeeds. Iff the variable is false, CAS sets it to true in an atomic operation. This condition is tested in a loop until the CAS operation was successful This is also called busy waiting.

In contrast to mutex locks, this does not require any expensive system calls. For locks with short contention time this may be an advantage in performance. A disadvantage is the single global Boolean variable required for the synchronization. Since the CAS instruction is a write operation, the cache needs to be exclusive.

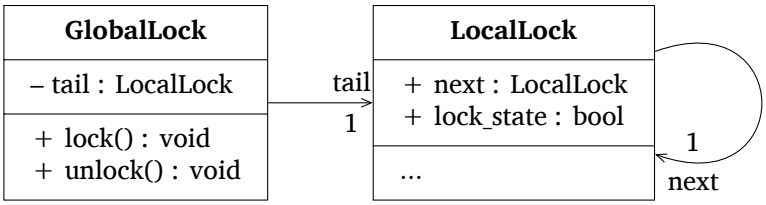


Figure 3.22: The UML class diagram of the global lock and local lock classes used for the MCS lock.

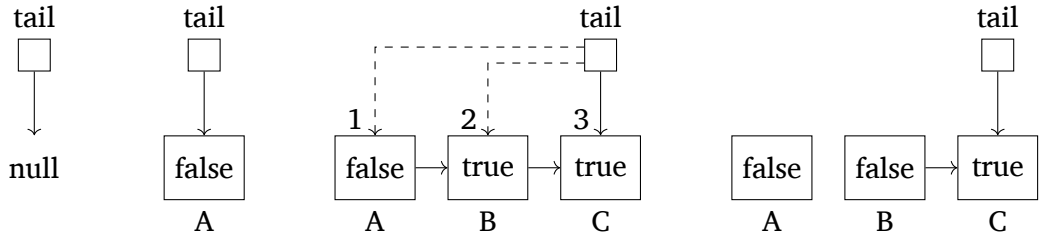


Figure 3.23: An example of an MCS-Lock adapted from [55, p. 154]. In the leftmost figure, the initial MCS-lock is shown. The second figure shows the state, after thread A acquired the lock. In the next figure, thread B and C want to acquire the lock as well. Therefore, the next pointer of the previous local lock is set to the own local lock and the own local lock state is set to true. The rightmost figure shows the state after thread A released the lock. In this figure, thread B holds the lock and thread C is still waiting.

This causes frequent cache invalidations. Especially, for multiple NUMA-nodes this causes overhead due to traffic on the bus connection.

Scalable Locks – MCS Locks

The MCS lock [78] is a so called scalable lock. It performs well even under high contention.

The MCS lock is designed as a FIFO lock, ensuring a kind of fairness for contending threads. The MCS lock is cache local and therefore cache coherence friendly due to local spinning instead of global spinning. However, the memory requirement is not constant, but scales with the number of threads.

Similar to Lamport's Bakery lock [74], the MCS lock uses a queue for maintaining threads trying to acquire the lock. The MCS lock consists of global lock and local lock objects (see Figure 3.22). The global part of the MCS lock consists of a pointer to the tail of a linked list representing the queue. The queue itself consists of local locks. Those local locks encompass a pointer to the next element in the queue and a Boolean variable representing the lock state. Initially, every thread using the MCS lock has its own local lock object initialized with a null pointer and a lock state set to false.

If the global lock's tail pointer is a null pointer, the lock is unlocked. The lock method uses an atomic fetch and store operation to set the global lock tail pointer to a pointer to its own local lock and fetches the predecessor. If the predecessor is

non-existent, the lock was unlocked and is locked by the thread. If a predecessor exists, another thread already holds the lock. The acquiring thread sets it own Boolean flag to true. Afterwards the acquiring thread sets the next pointer of the predecessor to its own local lock. Finally, the acquiring thread starts waiting on the local Boolean variable until the predecessor unlocks the lock.

The unlocking works as follows: The releasing thread checks the pointer of its own local lock to the next local lock. If this pointer is a null pointer, the releasing thread uses a compare and set operation on the global lock's tail pointer. It compares the tail pointer to the pointer of its own local lock and sets it to a null pointer. If this operation was successful, no other thread currently wants to acquire the lock and the lock is unlocked. If the compare and set fails, another thread is acquiring the lock and the releasing thread waits until the local next pointer was set by the acquiring thread.

If the next pointer of the local lock is not a null pointer anymore, the releasing thread sets the Boolean variable of the next local lock to false. This will release the lock and the next thread can acquire it. Additionally, the next pointer of the local lock of the releasing lock will be set to a null pointer.

Compared to a spin lock, all busy waiting is only done on local variables. The cache of another core is only modified once when the lock is released. This means, instead of multiple cache coherency invalidations only a single one is required.

3.5.4 Supporting NUMA

Up until now, only uniform memory access (UMA) hardware was considered. However, already today's hardware exhibits non-uniform memory access (NUMA). Due to NUMA, accessing data residing on remote NUMA nodes is significantly more expensive [83]. This needs to be considered for the task engine. In the task engine, the crossing of NUMA borders can happen during the work-stealing or the work-sharing.

A master thesis using the proposed task engine, investigating the effects of NUMA has been conducted by L. Morgenstern [85]. This work proposes NUMA aware strategies for work-stealing, load balancing and allocations of algorithmic data. It shows, that NUMA awareness is necessary, especially for latency-critical applications. Parts of the performance analysis of this thesis are shown in Section 3.7.4.

3.6 Use Case: FMM Parallelization

In Chapter 2 the algorithmic details of the FMM were presented. In this section this knowledge will be used to develop different parallelization strategies.

From the sequential flow of the algorithm likely synchronization points within the parallelization execution can be deduced. With this in mind, a first loop-level only parallelization can be formulated. This version would exhibit very coarse-grained synchronizations between the FMM passes.

To resolve those coarse-grained synchronizations the algorithmic dependencies will be analyzed in more detail resulting in the data-flow graph. This graph will

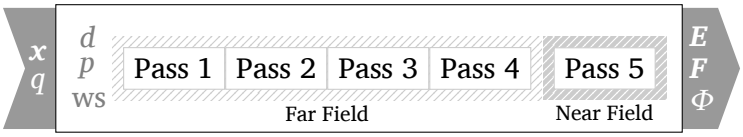


Figure 3.24: The sequential flow of the FMM subdivided in passes.

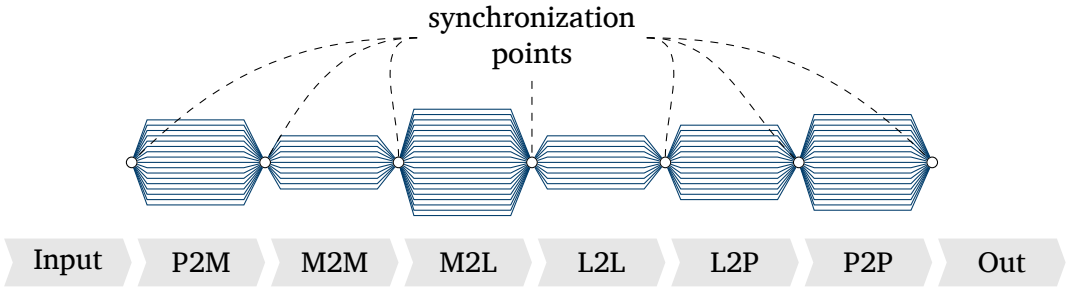


Figure 3.25: The loop-level parallel FMM using the loops of the passes for the parallelization. After each loop, an implicit synchronization is required.

be used for the data-flow parallelization which is the defined goal for the task engine design and implementation. It exhibits sufficient parallelism and promises the most scalability.

Sequential Algorithmic Flow

Figure 3.24 show the typical subdivision of the FMM into five different passes. Four passes are part of the far-field computations and one pass denotes the near-field computation. Before any pass can start, the space must be subdivided using an octree. Afterwards, the particles are binned into the boxes on the lowest level of the octree.

After those preliminary steps, the algorithmic flow starting with the far-field proceeds as follows: In *Pass 1* the particles are expanded into multipoles (P2M) and shifted and accumulated upwards in the tree (M2M). *Pass 2* consists of the translation of multipole expansions to local expansions (M2L). *Pass 3* encompasses the shift and accumulate operations downwards in the tree (L2L). *Pass 4* is the last pass in the far-field computation and comprises of the computation of the far-field forces affecting the particles (L2P). The computation concludes with *Pass 5*, the computation of the near-field forces (P2P).

Deduced from this algorithmic flow, a trivial parallelization could be implemented using the passes as synchronization points.

3.6.1 Classical Loop-Based Design

Every pass in the sequential FMM encompasses loops iterating horizontally or vertically in the tree. These loops can be parallelized using loop-level parallelization. Figure 3.25 shows the flow of a loop-level parallelized FMM. This approach is easy to implement and can be beneficial for nodes with only a few cores.

However, this approach does not strong-scale on nodes with a high number of

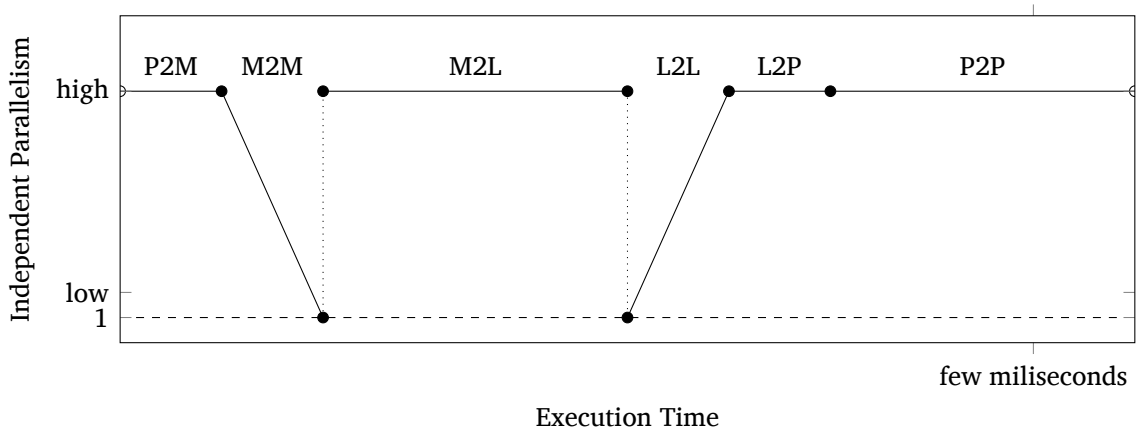


Figure 3.26: The FMM operators exhibit different amounts of parallelism. Operators like M2M and L2L have inter-operational dependencies and are limited in parallelism on higher tree levels. Operators working on one tree level do not suffer from this limitation and exhibit a higher level of parallelism.

cores with only a few particles per core. The limitations for the scaling are the following: Due to the parallelization over loops sequential regions outside the loop will remain. Additionally, the loop-level approach introduces unnecessary synchronizations. After each parallel loop, an implicit barrier causes the synchronization of all threads. This contributes to the overall sequential portion of the program. Corresponding to Amdahl's law [5] this will limit the speedup.

Another complication arises from the different amounts of work inside the different loops (see Figure 3.26). Operators without inter-operational dependencies like P2M, M2L, L2P and P2P are inherently independent and can be executed concurrently. All loop iterations are independent of each other and can therefore be executed in parallel as well. Operators like M2M and L2L have inter-operational dependencies. As an example, M2M shifts multipole expansions upwards in the tree. Hence, the most outer loop iterates over the tree levels starting at the lowest level. Before the upper level can be reached, the multipoles on the lower levels need to be shifted upwards. This reduces the parallelism from level to level up to a single box at the root node. This is the critical path for the parallelization, due to the lower parallelism on the higher tree levels. The runtime impact of these operations on the higher level might be small, but Amdahl's law will show degradation in the scaling if the number of cores becomes larger. This is an unnecessary bottleneck, which can be avoided by considering the critical path of the FMM for scheduling decisions and remove arbitrary synchronization points of the loop-level approach.

3.6.2 Evolution of a Data-Centric View

The trivial parallelization based on the sequential view is incoherently limited in scaling for the desired problems. To overcome disadvantages of the loop level parallelization the data-dependencies of the FMM need to be analyzed in more

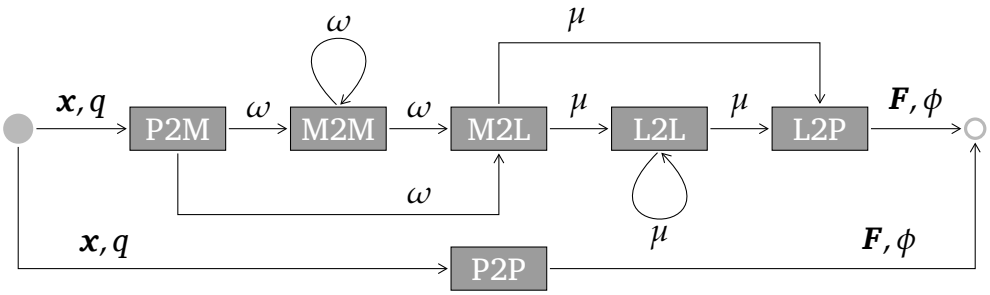


Figure 3.27: The data-flow graph of the FMM operators. The nodes represent the aforementioned operations. The edges denote the data-dependency between the operations.

detail. For the data-flow graph, the FMM specific operators and their input and output data must be analyzed. The data used in the FMM can be categorized as follows:

- particle coordinates and their charges,
- multipole expansions ω and local expansions μ and
- forces and potentials.

Figure 3.27 shows a simplified data-flow graph of the FMM. The boxes in this graph represent the operators, whereas the edges denote the in- and output data and therefore the dependencies. For simplicity, the spatial subdivision and the binning of particles into boxes is omitted in the shown graph.

The last work item in the algorithm is the computation of the forces. This step is finished, when both, the near-filed and the far-field forces are computed.

A loop-level parallel FMM would require synchronization points between the passes. This lowers the exhibited parallelism. With a parallelization utilizing the data-flow graph, these bulk synchronizations can be resolved and exchanged against several smaller individual synchronizations. In contrast to the synchronizations after the parallel loops, not all threads need to synchronize at those points. This will exhibit more independent parallelism at a fine-grained level, which makes increased concurrency possible.

3.6.3 Data-Driven Task Parallel FMM

The data-flow graph of the FMM is the basis of the data-driven task parallel FMM implementation. The data-flow graph is implemented using the static data-flow dispatcher described in Section 3.4. For the configuration of the dispatcher, the definition of tasks, data events and dependency counters are required.

The dependency counters are used to determine the state of the computation. They are used to decide whether the data is ready for further computation.

Figure 3.28: The M2L task can be source centric or target centric. In the source centric case, one multipole expansion ω is used for 189 M2L operations. In the target centric case, one local expansion μ will be computed using 189 M2L operations.

FMM Data-Events

The data-events for the FMM are determined by the data-flow graph. The input data (coordinates and charges) is not reflected as a data-event. It is expected, that this data is fully available at the beginning of the simulation. Nevertheless, if this is not the case it is possible to introduce a data-event for the input data as well.

For the multipole and local expansions different events are defined. The first event is called `OMEGA` and triggered, when a multipole expansion was computed. This is the case after a P2M operation for a box or eight M2M operations have been executed. The next event is called `MU`. This event is triggered whenever a local expansion is computed. This requires all 189 M2L operations and one L2L operation from the parent level to be finished. Finally, there is an event called `MU_Lowest` which is used for identifying local expansions on the lowest tree level. After the local expansion on the lowest level is computed, the subsequent L2P tasks are be created and enqueued.

Tasks Granularity

For the tasks an identifier is required. This is the box the operator will be applied on. Instead of creating several tasks of the same type for different indices, this can be done with a set of indices. This set of indices is iterated inside the task and the task function will be executed for each index. For simplicity the following description uses only one index instead of a set. Every FMM specific operator is implemented using a custom task type.

The tasks are defined as follows: The P2M task encompasses the expansion of all particles inside the box under consideration. After a single P2M task is executed, the multipole expansion for the indexed box is ready for further computation. The box index of an M2M task selects the parent box for the operation. The task consists of all eight M2M operations required for the computation of the multipole expansion in this box. It shifts all eight child box multipole expansions upwards.

The M2L operation can be modeled in two ways, either target centric or source centric (see Figure 3.28). For the computation of a local expansion ($ws = 1$) exactly 189 M2L operations are required. Due to the symmetry of the algorithm a multipole expansion is used in exactly 189 M2L operations as source. The number of M2L operations depends on the well separation criteria. This means, if one

multipole is used for all M2L interactions, the task is called source centric. If a task encompasses all M2L operations required for the computation of one target μ, it is called target centric.

For the L2L task the parent box index is used for distinction. The task consists of all downwards operations from this box to its eight child boxes. The L2P task is similar to the P2M task and encompasses all particles belonging to the box under consideration. The P2P task includes the computation in the near-field required for particles belonging to the box.

3.7 Performance Evaluation

In this section the performance of the proposed task engine using the data-driven FMM will be discussed. Referring to [109] the performance of the task engine will be analyzed in three different categories:

- Overhead-time
- Work-time
- Idle-time

The presented measurements are repeated sufficiently often to guarantee consistent results and 75 %-quartiles are plotted [58].

Particle Systems

The shown performance evaluations are conducted using the task engine for the FMM implementation. For all measurements the well separation criteria is set to $ws = 1$. The multipole order p is used to adjust the accuracy of the computation. A low multipole order reflects a low accuracy and leads to less computation, this moves the benchmark into the latency-critical regime. Low accuracy can be used to reveal additional but minor parallelization overheads.

The following two particle systems are used for the analysis: The first system is the *small* particle system. This system was generated randomly and encompasses 1000 homogeneously distributed charged particles. The system is used with a tree depth $d = 3$. This leads to 512 boxes on the lowest level encompassing two particles per box in average.

The second particle system is the *large* particle system. This system consists of 103 680 homogeneously distributed charged particles from a silica melt [7]. The tree depth for this example is set to $d = 4$.

Hardware Specification

The following analysis has been conducted on a compute node equipped with two Intel Xeon Platinum 8170 processors [63] encompassing 26 cores each. In the following this node will be called the *Skylake* node. Important characteristics of this system are summarized in Table 3.5. The system has two NUMA nodes and is equipped with 196 GiB of main memory.

System specifics		Caches	
Cores	26	L1d	32 KiB
Sockets/NUMA	2	L1i	32 KiB
Total cores	52	L2	1024 KiB
SMT	2	L3	36 608 KiB

Table 3.5: Important hardware characteristics of the used compute node equipped with two Intel Xeon Platinum 8170 CPUs and a total of 196 GiB main memory.

Additionally, some measurements have been conducted on a compute node of the supercomputer called JURECA [73]. These nodes are equipped with two Intel Xeon E5-2680 v3 processors [64] encompassing 12 cores each.

3.7.1 Overhead-Time Analysis

The measurement shown in Figure 3.29 was performed to reveal overheads introduced by the task engine. This analysis was done on the *Skylake* node using the *small* particle system and a multipole order $p = 0$. For 52 cores, the number of particles per core is less than 20. In the plots, the runtime and the parallel efficiency are shown. The low order of poles leads to tiny tasks encompassing only a single complex multiplication for each FMM operators M2M, M2L or L2L. It is not expected to achieve a high parallel efficiency for this corner case. In a real world example, the workload would be significantly larger. The plot still shows, with an increasing thread count, a decrease of runtime to 1.4 ms. Even with almost no computation at all, a parallel efficiency over 40 % has been achieved for 26 cores.

3.7.2 Work-Time Analysis

The analysis shown in Figure 3.30 focuses on the work-time per M2L step. Therefore, the numbers of M2L operations per second were measured depending on the number of threads used. This analysis was done on a node of JURECA using the *large* particle system and a multipole order $p = 10$.

For the analysis of the parallel execution overhead, the inflation of the work-time is interesting. Such an inflation could be due to different reasons, for example cache effects. The plot presented in Figure 3.30 reveals a much higher rate for one and two threads compared to 24 threads. However, this drop in performance can be explained fully by the dynamic frequency scheduling support of the processor, like Intel Turbo Boost [23] and does not hint towards scalability issues. Turbo boost allows the processor frequency to vary, depending on the overall workload (see Figure 3.31). With Turbo Boost enabled, no reliable measurements of the parallel efficiency are possible, since the base line version with only a single active thread might have been exposed to a faster CPU clock speed.

3.7.3 Idle-Time Analysis

The benchmark shown in Figure 3.32 is used for analysing the idle-time of the FMM using the task engine. This analysis was done on a node of JURECA using

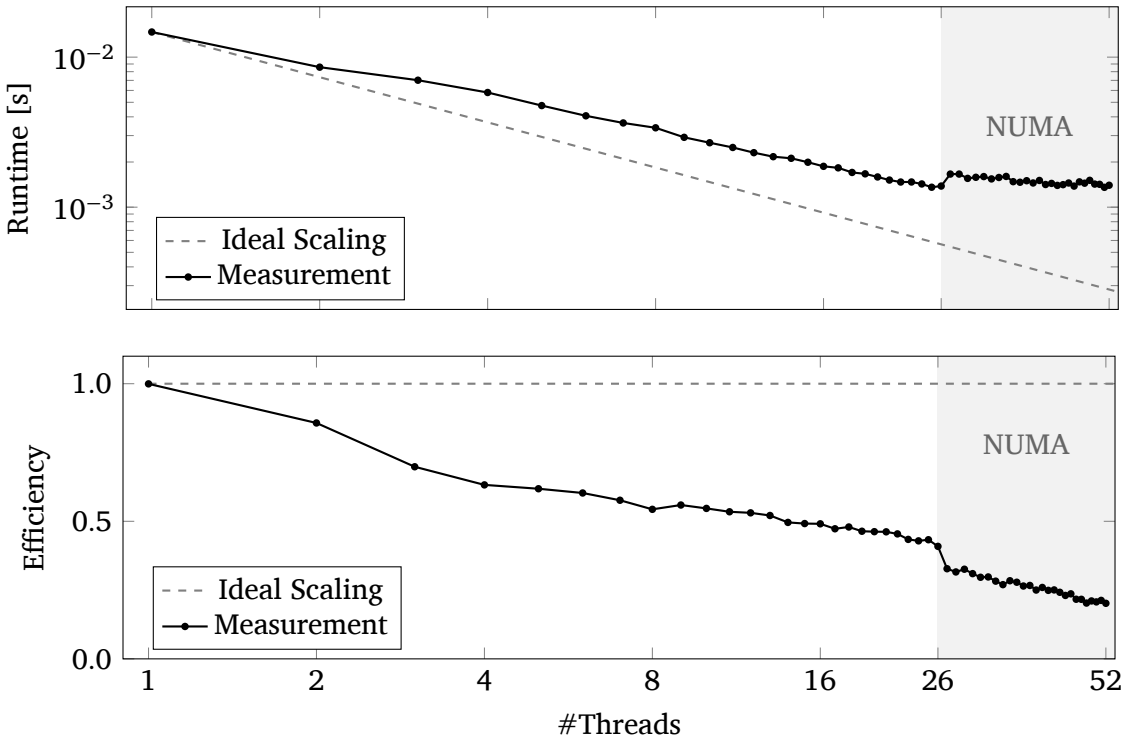


Figure 3.29: This benchmark uses the *small* particle system and a multipole order of zero. It was conducted on the *Skylake* node. These parameters reflect a very low accuracy resulting in almost no computation at all.

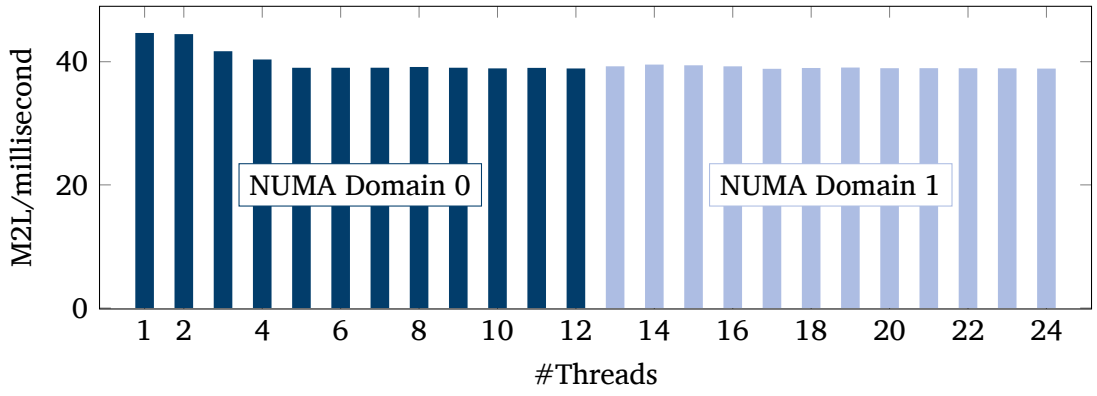


Figure 3.30: Work-time inflation for the M2L operator. Showing the number of operations per second depending on the number of threads. This benchmark uses the *large* particle system and $p = 10$. It was conducted on a single node of JURECA.

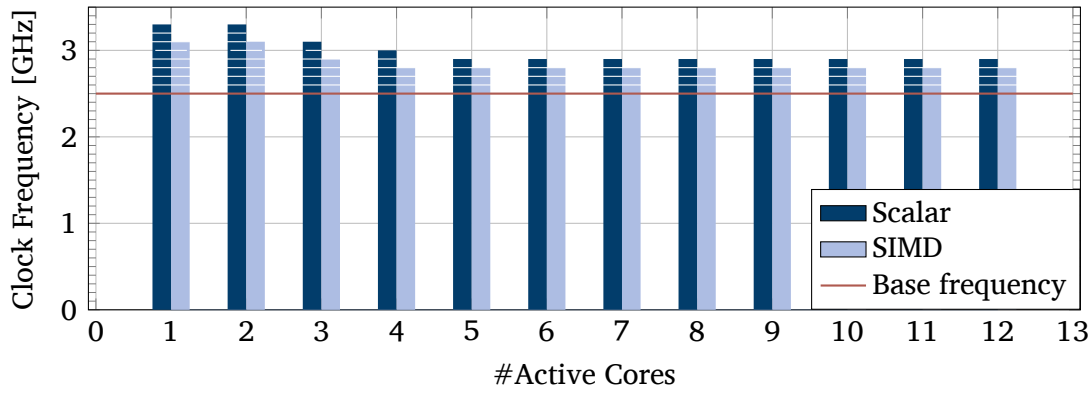


Figure 3.31: Intel Turbo Boost frequency bins for Intel Xeon E5-2680 v3 depending on the number of utilized cores.

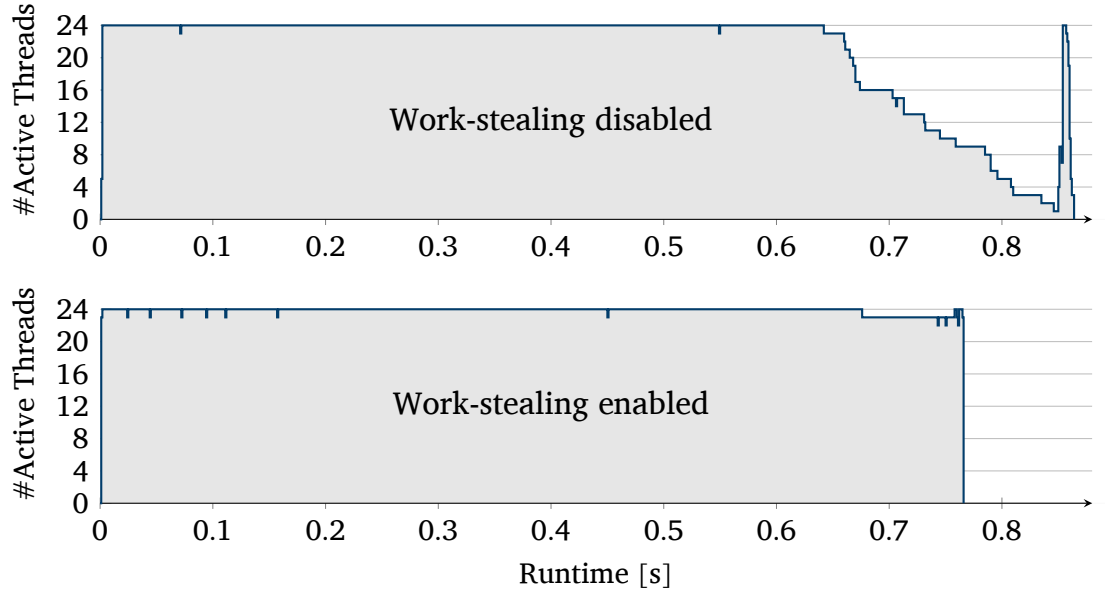


Figure 3.32: Plots showing the number of concurrently active threads for a simulation run with 24 threads. This benchmark uses the *large* particle system and $p = 10$. It was conducted on a node of JURECA encompassing 24 cores.

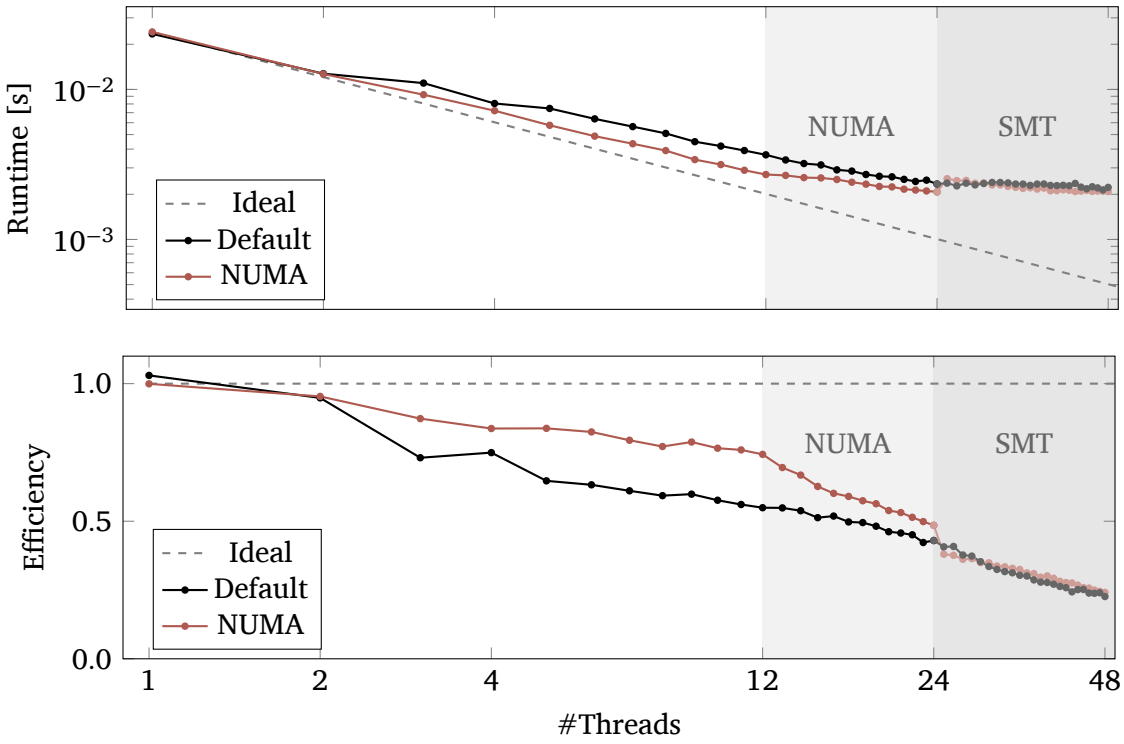


Figure 3.33: Comparison of the best NUMA-strategy and default implementation from [85]. This benchmark uses the *small* particle system and a multipole order of three. It was conducted on a node of JURECA encompassing 24 cores in two NUMA-nodes.

the *large* particle system and a multipole order $p = 10$. This benchmark was executed by 24 threads. During the computation, at every millisecond, the active threads were counted. Here, active thread means, that any thread computing a task and currently not being in the scheduling process counts as active. The benchmark with work-stealing disabled shows a drop of parallel active threads towards the end. This drop is due to the parallelization bottleneck and subsequent synchronizations in the upper tree levels. Some threads run out of work, while other threads are still shifting multipoles upwards in the tree. Only after the first L2L tasks can be created, every thread is actively participating again. As expected, this can be circumvented by dynamic scheduling using work-stealing.

3.7.4 NUMA-Awareness Analysis

Figure 3.33 shows the results of the NUMA-awareness extension described in more detail in the master thesis by L. Morgenstern [85]. This analysis was done on a node of JURECA using the *small* particle system and a multipole order $p = 3$. The employed FMM settings reflect a very low accuracy. This setting was chosen to introduce a higher runtime impact due to NUMA effects. In the plot, the baseline implementation without NUMA awareness is compared to the best NUMA strategy implemented. As shown, the NUMA aware implementation constantly outperforms the baseline implementation. This shows, NUMA already poses a performance

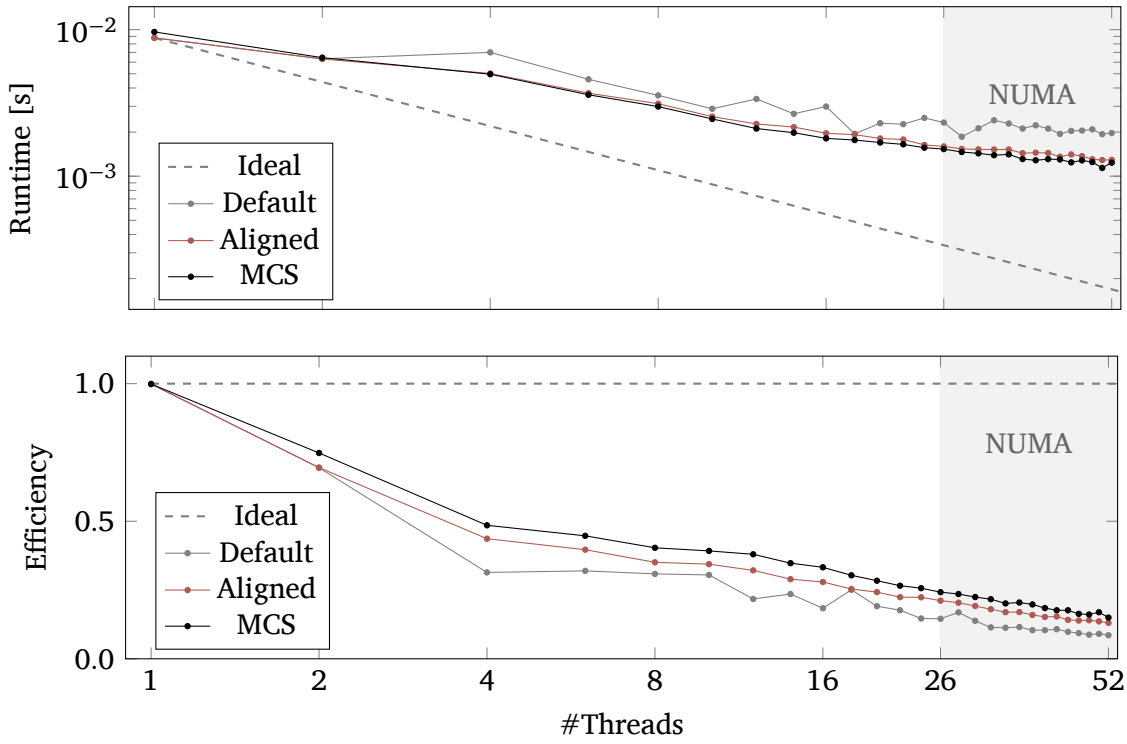


Figure 3.34: Comparison of mutex lock, a non-shared mutex lock and the MCS lock. This benchmark uses the *small* particle system and a multipole order of one. It was conducted on the *Skylake* node.

drawback and can be effectively handled with NUMA aware allocations, work-stealing and load balancing.

3.7.5 Lock Comparisons

The measurements shown in Figure 3.34 show a comparison of different lock implementations. This analysis was done on the *Skylake* node using the *small* particle system and a multipole order $p = 1$.

The *default* implementation uses mutex locks from the standard library. Since the locks are stored contiguously several locks will reside in the same cache line. This leads to false sharing of the locks. To avoid false sharing, the mutex locks have been aligned and implicitly padded to 64 B each (the length of the cache line) in the *aligned* plot. The *MCS* plot is using the MCS lock with aligned global locks. As seen in the plot, the alignment alone induces a high performance improvement. This improvement was as high as 40 % of runtime. But even between the aligned mutex locks and the MCS lock an improvement of up to 12 % can be achieved additionally. Most important, in all cases the aligned MCS lock outperforms the aligned mutex implementation and baseline implementation. This makes the MCS lock to the best choice as general purpose lock in the task engine.

One might argue to use a superior lock-free or wait-free implementation for the queue instead of any lock. However, lock-free implementations introduce

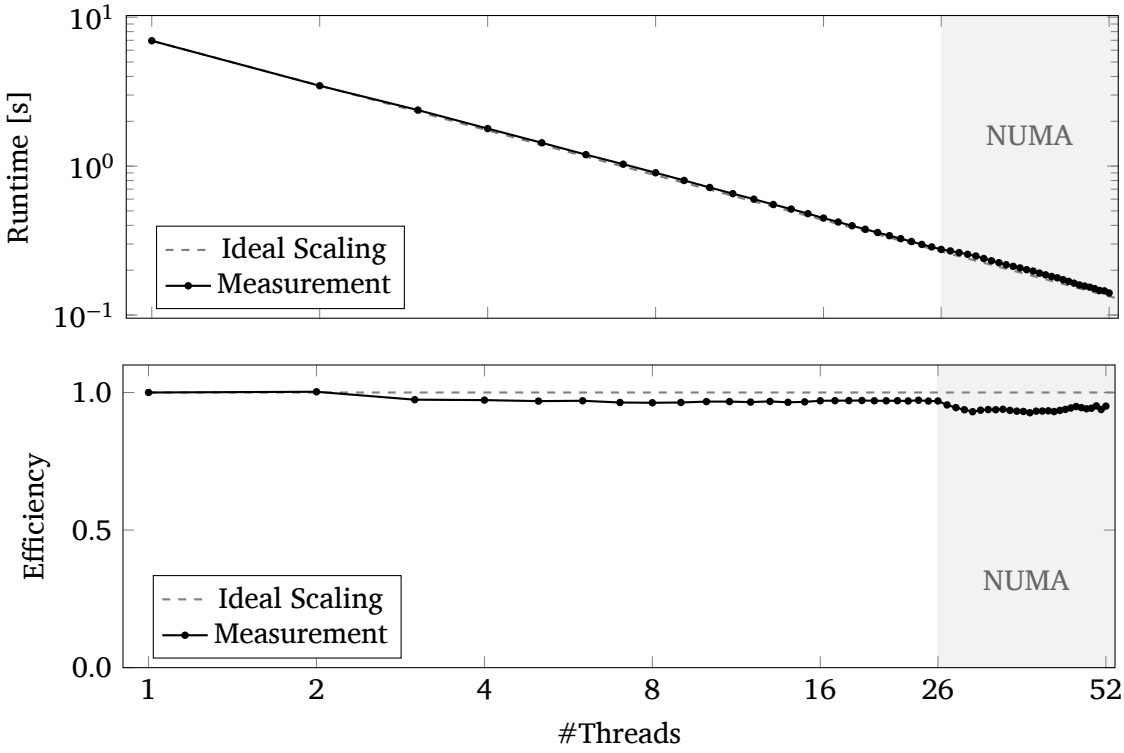


Figure 3.35: This benchmark uses the *large* particle system and a multipole order of 15. It was conducted on the *Skylake* node.

additional fine-grained synchronizations, which might impede the scaling. Additionally, every thread has several sub-queues reducing the contention further.

3.7.6 Real World Benchmarks

This benchmark uses a real world example using a reasonable multipole order $p = 15$ (see Figure 3.35). This analysis was done on the *Skylake* node using the *large* particle system. Using all 52 cores, approximate 2000 reside on a single core. The shown efficiency shows a consistently high performance up to 52 threads. For 26 cores a parallel efficiency of 96.9 % was achieved. Even when crossing the NUMA border no major performance drawback can be observed and the parallel efficiency for 52 cores is 95 %. The runtime using 52 cores is 140 ms.

Low Accuracy

This benchmark uses the same example as the previous benchmark but with a lower multipole order (see Figure 3.36). This analysis was done on the *Skylake* node using the *large* particle system and a multipole order $p = 5$. The lower multipole order leads to lower accuracy and thereby lower computational efforts. Nevertheless, the benchmark reveals still a good overall performance. Using 26 cores a parallel efficiency of 92.6 % can be achieved. For 52 cores the parallel efficiency is 87.7 % and the runtime is 43.3 ms.

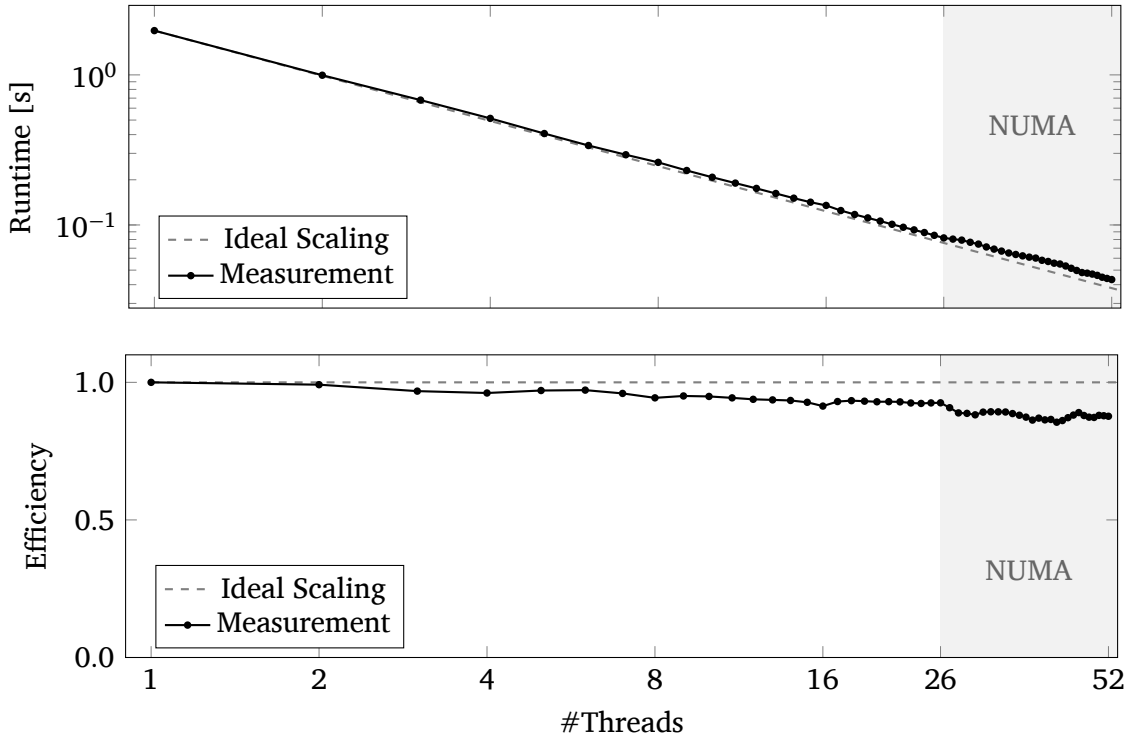


Figure 3.36: This benchmark uses the *large* particle system and a multipole order of 5. It was conducted on the *Skylake* node.

CHAPTER 4

A C++ Task Engine: Inter-Node Extension

Until now, the proposed task engine was discussed with shared memory capabilities in mind. For the parallelization on supercomputers inter-node parallelization features are inevitable. Thus, the extension of the task engine towards inter-node communication will be explained in this chapter.

Inter-node parallelization is not new in the field of high performance computing. Already the first massively parallel supercomputers consisted of several nodes and only a few cores per node. For those supercomputers, inter-node parallelization was already required whereas intra-node parallelization was not necessary.

In contrast to shared memory parallelization, inter-node parallelization requires explicit data transfer via messages from one node to another. This can be done with the concept of message passing. One effort of standardizing the message passing approach is visible in the message passing interface (MPI) standard [43]. Nowadays, implementations of the MPI standard are the de-facto standard for inter-node communication in HPC [72].

The current version (3.1) of the MPI standard does not provide a C++ interface. However, it is possible to directly use the C interface from C++. Independent of the existence of a C or C++ interface, additional abstractions are necessary. This has two main reasons: the first is the required separation of concerns and the second is the simplification of the interface for the user. Separation of concerns is important in order to stay flexible with respect to the underlying communication library. For inter-node parallelization the task engine uses message passing. This is not necessarily restricted to MPI and should be exchangeable for other libraries supporting point-to-point communication. Directly using MPI functions throughout the implementation would impede such an option.

Secondly, the simplification of the interface is needed since MPI requires several parameters in a function call which could be inferred from the message data automatically or even setup with default values. An early implementation of a C++ MPI interface is provided by Boost.MPI [51]. Unfortunately Boost.MPI is only an object oriented version of the MPI interface and does not hide all redun-

dant parameters. Additionally, Boost.MPI is based purely on MPI, which would require additional abstractions if other communication libraries should be supported. Another approach that drastically simplifies the user interface is provided by "MPP: An MPI CPP Interface" by Pellegrini [95]. This library contains many good thoughts on simplifying the interface using stream-like communication endpoints and stream operators on those endpoints. However, MPP only implements point-to-point communication and the development is stalled since 2013. Nevertheless, some features of MPP can be recycled and extended for the proposed C++ communication layer in this work.

A completely different approach to handle inter-node parallelization is the so called partitioned global address space (PGAS) approach, going beyond the aforementioned simplification of the interface. This approach is implemented by various libraries like HPX [70] or DASH [47] and language extensions like Co-array Fortran [90], Charm++ [71] or UPC [48]. For the PGAS approach new data-structures are provided behaving like normal non-distributed containers. These data-structures are distributed among nodes and handle the data access automatically. For the user of those libraries, the new data-structures behave like local containers. Whenever remote data is accessed, the retrieval is done by the library using its underlying communication capabilities. Typically, but not necessarily, PGAS implementations improve the performance by using one-sided communication via remote direct memory access (RDMA) [6].

This approach however has a drawback. The simple usability of those already familiar data-structures lead to an underestimation of the performance costs by the user. Since the distributed containers work similar to local data containers, the user will be tempted to use them alike. At the same time, the costs of remote data-access is in the range of microseconds which is 1000-fold higher than the local access costs which is only in the range of nanoseconds. From a performance point of view, this makes it too easy-to-use the interface incorrectly. Additionally, this does not satisfy an important rule of user interface design: "Make interfaces easy-to-use correctly and hard to use incorrectly." [81].

Especially for latency-critical application, the cost of remote data-retrieval is substantial. To efficiently parallelize latency-critical applications explicit communication across node boundaries must be predefined by the user. This allows for communication hiding or latency avoiding techniques in the message passing layer. Hence, for latency-critical application the PGAS approach is too susceptible for performance losses, since it cannot provide the same access to the communication methods as explicit communication. As a clarification it should be mentioned, that automation is an advantage for any library. Also the communication layer proposed in this work automates parts of the communication. Nevertheless, this automation is done without raising the expectation of a purely local data-access.

In the communication layer of this work, the communication remains visible and explicit. Communicating distributed data in the getter method (like the `[]` operator) of a container hides this performance penalty in an unfavorable way. The interface of a library should not hide performance critical parts, unexpected

by the user.

The communication between nodes using only node-local task engines imposes another challenge – specific to task engines for message passing. Historically, the point-to-point communication worked via bulk synchronous parallelism [113]. This means, data was computed in one phase and was communicated in another phase after the computation. However, with a task engine as proposed in this work the main synchronization points are dissolved due to the data-driven design. This also means, that these now non-existing synchronization points cannot be used for the communication anymore. Another pitfall is introduced by very tiny tasks modifying only small amounts of data. Sending several small messages directly after such a task is finished is not optimal. Small messages are dominated by the network latency and will introduce a performance penalty. This means, the data should be collected on the node and sent in larger chunks after a sufficient amount of data is ready for communication.

4.1 The Message Passing Interface (MPI)

The message passing interface (MPI) is a standard for inter-node communication. It defines the semantics and syntax of communication functionalities. It was developed by academia and industry to standardize the programming of message-passing programs. As of today, there are two main competitors for MPI implementations: MPICH [86] and OpenMPI [92]. Several other MPI implementations and especially vendor implementations exist but are based on one or the other.

History of MPI

The first version of the standard [42] consists of 129 functions and was published in 1994. It includes point-to-point communication as well as collective communication. It also includes language bindings for C and Fortran77, but not for C++. Version 2.0 was published in 1997 and includes 221 functions. Process management, parallel I/O capabilities as well as one-sided communication had been added to the standard. Additionally, new language bindings for C++ [105] had been introduced with version 2.0.

In 2012 the significantly increased version 3.0 of the standard was published

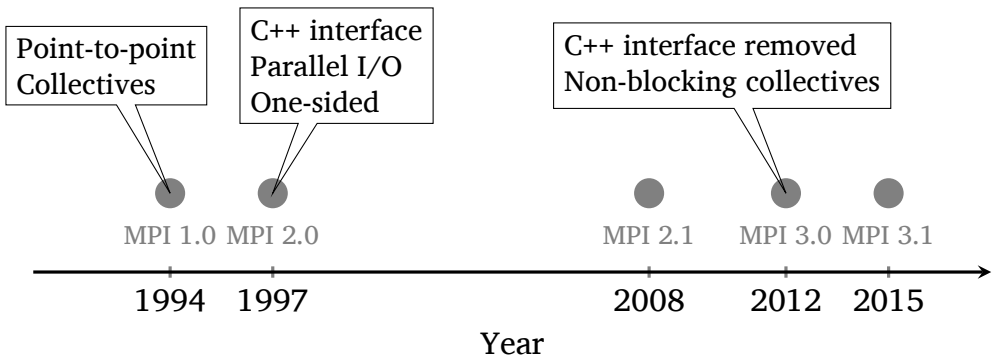


Figure 4.1: Historical development of MPI versions and major features.

Figure 4.2: Schematic view a single point-to-point communication via MPI. Sender rank 0 sends 10 consecutive doubles to receiver rank 1.

LISTING 4.1: REFERENCE OF NON-BLOCKING SEND AND RECEIVE

```
1  int MPI_Isend(const void * buf,
2                int count,
3                MPI_Datatype datatype,
4                int dest,
5                int tag,
6                MPI_Comm comm,
7                MPI_Request * request);
8
9  int MPI_Irecv(void * buf,
10               int count,
11               MPI_Datatype datatype,
12               int source,
13               int tag,
14               MPI_Comm comm,
15               MPI_Request * request);
```

including 443 functions. Especially non-blocking collectives and new one-sided communication features like RDMA support had been added. Besides this, the C++ language bindings was removed again. With minor changes, Version 3.1 [43] is the most current MPI standard available today.

4.1.1 BASIC MPI COMMUNICATIONS

In the following, the semantics of certain MPI communication functions will be discussed. For the task engine mainly two-sided communication and collectives are used. The syntax is shown for a few selected methods only. For a full list of MPI functions and features, the standard for version 3.1 [43] of MPI can be consulted.

POINT-TO-POINT COMMUNICATION

Point-to-Point communication is an elemental communication approach. It involves two communicating MPI processes called ranks. One is the sending process and the other one is the receiving process. Both processes are required to call the corresponding send and receive functions. The sending process provides the data to be sent and the receiving process provides a sufficiently large block of memory to receive the data.

The send and receive functions can be called in a blocking or non-blocking fashion. In the following the non-blocking send and receive calls are explained

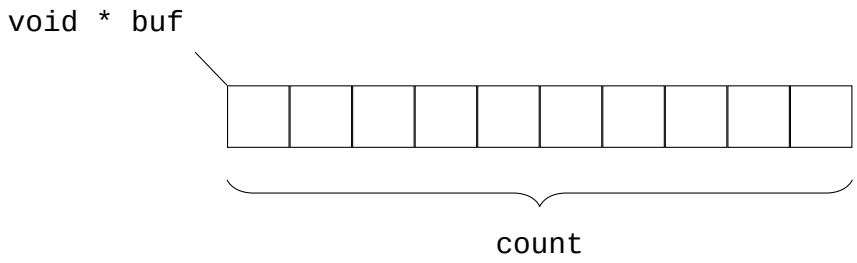


Figure 4.3: Layout of the data required by MPI for sending and receiving.

in more details: Listing 4.1 shows the interface of the non-blocking send and the non-blocking receive functions. Both methods require message data specific parameters (see Figure 4.3): a void pointer to a buffer, a count and an MPI datatype. For the sending process, this reflects the data which should be sent and for the receiving process this defines the memory allocation for the data received. For both functions, the void pointer called `buf` points to the first element of the data. The count variable represents the number of elements in this buffer and the MPI datatype is the internal MPI type of an element of the data. In the case of send, the buffer might be constant, since it this is a read only operation. This constant correctness for the C interface was introduced with the MPI 3.0 standard leading to a slight inconsistency between different MPI versions. Next, send and receive require the arguments which are part of the so called message envelope: the source or destination, the tag and the communicator. The message envelope is used for identifying matching send and receives. Additionally, for the non-blocking methods, an `MPI_Request` pointer must be added as the last function parameter. This request object holds information about the communication itself. After a non-blocking send or receive was emitted, the communication might not be finished right away and the request must be used for testing or waiting on this specific communication handle using `MPI_Test` or `MPI_Wait`.

Collective Communication

In contrast to point-to-point communication, collective communication involves all MPI processes belonging to a communicator or group of ranks. Examples of collective communications are gather, allgather, broadcast and alltoall. As an example only allgather should be explained in the following. A gather collects data from all participating processors in the communicator. An allgather additionally distributes this gathered data back to all processors within the same communicator.

Listing 4.2 shows the allgather function reference from the MPI standard. All MPI processes (ranks) that are part of the communicator must emit this function call. The data which should be sent in this communication is denoted by the void pointer to the send buffer, the send count and the send MPI datatype. As in the `MPI_Isend` function shown before, the send buffer can be provided const qualified. The receiving data is denoted by the receiving buffer pointer, the receiving count and the receiving MPI datatype. Additionally, the communicator for collectives

Listing 4.2: Allgather collective communication from MPI standard

```
1 int MPI_Allgather(const void * sendbuf,
2                   int sendcount,
3                   MPI_Datatype sendtype,
4                   void * recvbuf,
5                   int recvcount,
6                   MPI_Datatype recvtype,
7                   MPI_Comm comm)
```

Listing 4.3: Example of a generic 3D coordinate class

```
1 template <typename value_type>
2 struct XYZ {
3   value_type x, y, z;
4 };
```

must be set. In this case, the send count denotes the number of data sent in the allgather. The receive count denotes the total number of elements received.

MPI Datatypes

For the communication MPI uses MPI datatypes internally. These datatypes correspond to the distinct type used in the communication. For heterogeneous platforms this feature helps converting the data to the correct format, e.g. little-endian and big-endian for floating point numbers. MPI offers several intrinsic datatypes like `MPI_DOUBLE` or `MPI_INT`. For more complex or composite datatypes, MPI offers so called derived datatypes like `MPI_Type_contiguous`. These derived datatypes are composed from basic datatypes using MPI datatype constructors. With `MPI_Type_contiguous`, datatypes consisting of contiguous elements of the same type can be combined into a new datatype. For the communication, MPI analyzes the datatype and passes through the memory correspondingly. In contrast, derived datatypes do not need to be contiguous in memory. Internally MPI represents the datatype using two sequences, a sequence of basic datatypes and a sequence of displacements. This allows arbitrary datatype definitions.

Example of Point-to-Point Communication

The following example is used to illustrate basic point-to-point communication with MPI. It will be used as basis for later simplification and abstraction towards a high level user interface. For a realistic communication example a generic three-dimensional coordinate is used (see Listing 4.3). It consists of three member variables of the type `value_type`. These could be float or double. In the following, objects of this class should be sent and received using point-to-point communication.

The send and receive example shown in Listing 4.4 uses a vector containing 10 coordinate objects. The object was initialized for coordinates of type double. At first, a derived MPI datatype needs to be created (at line 4). Since the elements are contiguous in memory, a contiguous MPI datatype with three contiguous elements of type `MPI_DOUBLE` is used.

LISTING 4.4: EXAMPLE OF POINT-TO-POINT COMMUNICATION OF XYZ

```
1  std::vector<XYZ<double>> send_coordinates(10), recv_coordinates(10);
2
3  // defining the data type
4  MPI_Datatype datatype = MPI_DATATYPE_NULL;
5  MPI_Type_contiguous(3, MPI_DOUBLE, &datatype);
6  MPI_Type_commit(&datatype);
7
8  int my_rank, num_ranks;
9  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
10 MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);
11
12 // recv from +1 and send to -1
13 int recv_from = (my_rank + 1) % num_ranks;
14 int send_to = (my_rank - 1 + num_ranks) % num_ranks;
15
16 // send the data (blocking)
17 MPI_Send(send_coordinates.data(),
18          send_coordinates.size(),
19          datatype,
20          send_to,
21          0,
22          MPI_COMM_WORLD);
23
24 MPI_Status status;
25 MPI_Recv(recv_coordinates.data(),
26          recv_coordinates.size(),
27          datatype,
28          recv_from,
29          0,
30          MPI_COMM_WORLD,
31          &status);
```

Afterwards, basic information of the communicator are requested, like local rank id and the size of the communicator (at line 8). The rank id of the sending and the receiving process are calculated from this information. The shown communication scheme will send to the “left” (-1) and receive from the “right” ($+1$).

After these preliminary steps, the call to MPI send will be executed on line 17. MPI requires that the data used in the communication resides in contiguous memory. Since the implementation of `std::vector` is guarantied to store the data contiguously, it can be used directly for sending and receiving. Thus the `data()` method of the vector returns a pointer to the first element in the vector. The next parameter `count` provides the length of the vector. The datatype and destination rank have been created beforehand and are now used in the function call. For this communication no special tag was used and the default value zero was applied. The last parameter represents the communicator. Since for this example no special communicator is used, this is set to the default `MPI_COMM_WORLD`. The receive call works similar (at line 25), except that an additional status object is added as last parameter.

In this example each MPI process will hold two coordinate vectors. The first vector is named `send_coordinates` and is sent to the “left” neighbor. The second vector is named `recv_coordinates` and will be used for receiving the coordinates of the “right” neighbor.

4.1.2 Multithreaded MPI

Up until now, each rank only consists of a single MPI process and no additional threads spawned by the user existed. To allow messages to be issued from other threads and not the main MPI process, MPI needs to be made aware of this fact. Using MPI in a multithreaded fashion is still a recent topic of research and needs to be handled carefully because of the additional locking of shared memory done inside MPI. For a small number of cores per node it has been sufficient to ignore threading entirely and use multiple MPI processes (ranks) per node. However, with an increasing number of cores per node the overhead due to multiple MPI processes on a single node will increase. Especially in settings with a large number of nodes and hundreds of cores per node, the synchronization of collective operations will be costly. Additionally, the penalty of forgoing shared memory advantages for exchanging data on the node is a substantial performance drawback. These reasons make it necessary to take a closer look at multi-threaded MPI.

The MPI standard itself provides the following thread-safety modes:

`MPI_THREAD_SINGLE` The process is not multithreaded.

`MPI_THREAD_FUNNELED` The process can be multithreaded but only one thread is allowed to emit MPI function calls.

`MPI_THREAD_SERIALIZED` The process can be multithreaded and all threads are allowed to emit MPI function calls, but not concurrently.

`MPI_THREAD_MULTIPLE` The process can be multithreaded and the use of MPI is not restricted.

Despite the modes described in the standard an implementation is only required to implement `MPI_THREAD_SINGLE`. Also it should be noted that, current multithreaded MPI implementations do not always improve the performance when used with multiple threads. Rather the opposite is true, multithreaded MPI introduces a performance drawback [52, 110]. Therefore it could be beneficial to implement different communication models like communicating using a single thread for the communication in funneled mode. This requires the task engine to allow flexibility and also requires it to adapt the model available and best supported by the MPI implementation.

4.1.3 Shortcomings of MPI

The MPI standard increased significantly over time with more than 400 functions in the latest standard (3.1). For the communication layer proposed in this work, only a few functions of the MPI standard will be used. These functions are mainly two-sided point-to-point functions and a few collective communications functions. This means, more than 90 % of the standard will be ignored for this implementation.

The size of the standard also became a burden for MPI itself [104]. Adapting new hardware features while maintaining backward compatibility is almost impossible.

Additionally, MPI is supposed to be used by application developers as well as library developers. This requires compromises for both sides and leads to a standard which does not fit either [36].

Another shortcoming can be found in the message matching of MPI performed for the receiving of messages internally. Message matching is an essential feature, since a high ratio of the communication time is spent in it [72]. The message matching is utilizing the message envelope containing source, tag and communicator. Since the MPI standard allows wildcards to be used for the source and the tag (`MPI_ANY_TAG` or `MPI_ANY_SOURCE`) the message matching is impeded. To realize wildcard-based message matching two queues are required internally. One for expected messages (the receive function was called already) and one for unexpected messages (no receive function called yet). If a new message arrives, the expected message queue will be traversed and a matching receive is searched. If no matching receive can be found, the message will end up in the unexpected message queue.

If an upcoming MPI standard denies the capability of using wildcards in the message envelope, namely any source or any tag, the implementation of the message matching could be done faster [30]. Since the tag and source are distinct, the message envelope can be hashed and therefore hash-tables can be used instead of queues. In contrast to the average complexity of $\mathcal{O}(n)$ for searching an element in a queue, a hash-table provides $\mathcal{O}(1)$ complexity. Reducing the time spent for the message matching will particularly decrease the latency of receiving a message which is of major importance for latency-critical applications.

For the proposed communication layer these shortcomings mean, that the implementation needs to be independent from the underlying communication library. MPI can be used as a default library supported on the most supercomputers but it should be possible to exchange MPI with other low level communication libraries.

4.2 A C++ Communication Layer

In this section the communication layer of the task engine will be introduced. The discussion of MPI yields two requirements for the communication layer:

- the simplification of the high level user interface and
- the separation of concerns.

The simplified user interface should relieve the algorithm developer from setting redundant parameters in the communication process. Especially parameters that could either be deduced from the program (like datatypes) or set to reasonable default values (like communicator or tag). An ideal point-to-point communication would only require to set the corresponding sender or receiver and the data to be transferred itself. Everything else, like serialization, datatype resolving and counting should be performed by the communication layer.

LISTING 4.5: MPITYPETRAITS TEMPLATE PRIMARY DEFINITION

```
1 // Primary class template
2 template <class T>
3 struct MpiTypeTraits {
4   static MPI_Datatype GetType() {
5     return MPI_DATATYPE_NULL
6   }
7 };
```

LISTING 4.6: MPITYPETRAITS SPECIALIZATION FOR INT

```
1 // Class template sepcialization for int
2 template <>
3 struct MpiTypeTraits<int> {
4   static MPI_Datatype GetType() {
5     return MPI_INT;
6   }
7 };
```

4.2.1 MPI TYPE TRAITS

As mentioned before, MPI uses its own internal datatypes for the communication. C++ is a typed language and therefore it is not necessary to declare the datatype manually. This can be done automatically by using TMP type traits mapping the C++ type to the corresponding MPI datatype.

Listing 4.5 shows the primary definition of the class template `MpiTypeTraits`. The type trait class is used to retrieve the MPI datatype directly from the data communicated. This class has a template parameter `T` reflecting the C++ type of the data. Additionally, the class has a static function `GetType` returning the corresponding MPI datatype. This means, the type `T` is mapped to the corresponding MPI datatype using the `GetType` method. The primary template also defines the default MPI datatype, which is set to `MPI_DATATYPE_NULL`. To provide a new mapping between a C++ and an MPI datatype, this class template needs to be specialized for this type.

Listing 4.6 shows the specialization for the `int` type. This basic datatype is mapped to the MPI datatype `MPI_INT`. For all basic MPI datatypes this works correspondingly.

LISTING 4.7: DEFINING A GENERIC TYPE FOR A 3D COORDINATE

```
1 template <typename SUBT>
2 struct MpiTypeTraits<XYZ<SUBT>> {
3   static MPI_Datatype GetType() {
4     static MPI_Datatype type = MPI_DATATYPE_NULL;
5     if (type == MPI_DATATYPE_NULL) {
6       MPI_Type_contiguous(3, MpiTypeTraits<SUBT>::GetType(), &type);
7       MPI_Type_commit(&type);
8     }
9     return type;
10  };
11 };
```

LISTING 4.8: REMOVE THE CONSTANT QUALIFIER FROM THE TYPE

```
1 // Remove const from type, not needed for MPI_Datatype
2 template <typename T>
3 struct MpiTypeTraits<const T> {
4   static MPI_Datatype GetType() {
5     return MpiTypeTraits<T>::GetType();
6   }
7 };
```

Additionally, type traits can be nested and thereby used for the creation of derived datatypes. The three-dimensional coordinate shown in Listing 4.3 is an example of a contiguous datatype. It consists of three contiguous elements of the generic type `value_type`. Since this value type can be used with different concrete types, the type trait must be generic as well. For this purpose the type trait class must be partially specialized (see Listing 4.7) using the template parameter `SUBT`. The shown type trait is valid for all three-dimensional coordinates using the value type `SUBT`.

The static `GetType` method has an additional static variable called `type`. This is initialized once and set to `MPI_DATATYPE_NULL`. Since this is a static function variable, every subsequent call to this method uses the same variable. The actual MPI datatype is constructed using the datatype constructor `MPI_Type_contiguous` for three contiguous elements. The datatype of the element can be retrieved by using the type traits class itself and by resolving the corresponding MPI datatype for `SUBT`.

However, these C++ type traits hold a problem. Since C++ types differ for not qualified of const qualified types, a separate type for `int` and `const int` is required. A tedious over-definition of types could be solved by nesting type traits as well (see Listing 4.8). This type trait specialization maps a type `const T` and resolves it by calling the type trait class for the type `T`.

The nested type trait functionality is very powerful but unfortunately has its own design flaw. The implementation shown here is not thread-safe. In the case of multithreaded communication, a mutex or atomic flag must be used for the synchronization. The thread starting the registration must be the only thread registering the type. All other threads need to wait until the type is full registered.

4.2.2 SERIALIZATION

MPI datatypes are not always the best solution for sending data. For some datatypes it is beneficial to use serialization (a stream of bytes without a type) instead of MPI datatypes for the communication. This can have two different reasons: The first reason is, that compiled serialization is faster than dynamic interpretation of MPI datatypes [104, pp. 6]. The second reason is, that some complex C++ data structures cannot be mapped to MPI datatypes (e.g. objects containing dynamic allocations) in a reusable fashion. Therefore, an additional serialization infrastructure is required besides the aforementioned MPI type traits.

Since the algorithm developer knows the algorithm-specific data in depth, the

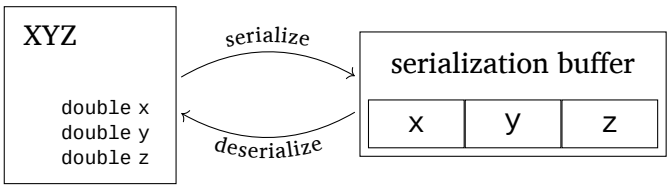


Figure 4.4: Schematic flow of the serialization and deserialization of the three-dimensional coordinate.

serialization methods of an object must be provided by the algorithm developer. Intuitively serialization requires three methods: one for serialization, one for deserialization and another for calculating the required memory size of the serialized object. As shown in the following example, this can be reduced to a single method which provides the same serialization capabilities. This makes the interface for the high level algorithm developer much simpler and reduces unnecessary code duplication. The described interface is adapted from the Boost serialization library [99] and extended to support nested data structures.

As an example the three-dimensional coordinate from Listing 4.3 should be serialized. For the serialization a serialization buffer is required. To allocate an appropriately sized buffer, the serialized size of the three dimensional coordinate needs to be calculated. Then, the serialization itself is performed by copying the coordinates into the serialization buffer (see Figure 4.4). The order in which the elements of the object are copied into the serialization buffer is not important. It is only important, that the order for serialization is the same as for the deserialization. The deserialization reinterprets the bytes in the serialization buffer and stores the floating point values in the coordinates (`x,y,z`) again.

The important finding from this example is, that the serialization and deserialization of an object is basically the same. For the serialization data is copied from an object into the serialization buffer and for the deserialization the data is copied out of the serialization buffer into the object. This means, it must be sufficient to require a single serialization method to be implemented by the user. Then, this scheme will be used for both serialization and deserialization. Additionally, the serialization tools proposed in this work will determine the serialized size required by the object automatically.

The serialization is implemented using different overloads of the ampersand operator (`&`) depending on different serialization buffer adapters. These overloads are used internally and no additional implementations by the algorithm developer are required. In the following the serialization of a single object is explained in more detail. The serialization of several objects of the same type in a vector or array is performed by a data wrapper by iterating and serializing each object separately.

The High Level Algorithm Developer's View

Let's discuss the required serialize method for the three-dimensional coordinates example (see Listing 4.9). The signature of the `serialize` method is predefined by

LISTING 4.9: THE SERIALIZE METHOD OF THE 3D COORDINATE

```
1  struct XYZ {
2    double x, y, z;
3
4    template <typename SerializationAdapter>
5    void serialize(serial::SerializationBuffer<SerializationAdapter> & s_buf) {
6      s_buf & x;
7      s_buf & y;
8      s_buf & z;
9    }
10 }
```

LISTING 4.10: THE SERIALIZE METHOD OF A NESTED OBJECT

```
1  struct XYZq {
2    XYZ<double> coordinates;
3    double q;
4
5    template <typename SerializationAdapter>
6    void serialize(serial::SerializationBuffer<SerializationAdapter> & s_buf) {
7      s_buf & q;
8      s_buf & coordinates;
9    }
10 }
```

the serialization library. It encompasses one template parameter defining the serialization adapter. The single function parameter is a serialization buffer passed by reference using the serialization adapter from the template parameter. The function body defines the serialization and deserialization order using the ampersand operator (`&`). In this example the order of serialization and deserialization is first x, then y and finally z. The serialization will start copying the x coordinate into the serialization buffer, afterwards the y coordinate and lastly the z coordinate. This order is completely arbitrary and can be changed by the user since the deserialization copies the values from the serialization buffer into the object using the exact same order.

Additionally, the ampersand operator supports nested data structures. This can be used for more complicated objects consisting of sub-objects. In this case, the algorithm developer starts by defining the serialize methods for the sub-objects. Afterwards the serialize method of the composed object can use the ampersand operator to define the serialization order of the sub-objects. Listing 4.10 shows an example of a class using the three-dimensional coordinate and an additional charge `q`. It should be mentioned again, that the order of serialization does not need to be the same as in the class definition.

THE LOW LEVEL LIBRARY DEVELOPER'S VIEW

The high level user interface has been presented, but the design of the internal structure requires further explanation. The signature of the serialize method (see Listing 4.9) is fixed to a single function parameter. This single function parameter is the serialization buffer. The serialization buffer can be used with different

LISTING 4.11: SERIALIZATIONBUFFER AMPERSAND OPERATOR IMPLEMENTATION

```
// Call the nested serialization for serializeable types.
template <typename value_type>
typename std::enable_if<IsSerializeable<value_type>::value, void>::type
operator&(value_type & value) {
  value.serialize(*this);
}

// value_type is not serializable so adapter can be applied.
template <typename value_type>
typename std::enable_if<!IsSerializeable<value_type>::value, void>::type
operator&(value_type & value) {
  SerializationAdapter::Apply(value, storage_);
}
```

LISTING 4.12: PARTS OF THE INPUTSERIALIZATIONADAPTER

```
struct InputSerializationAdapter {
  template <typename value_type>
  static void Apply(const value_type & value, storage_impl & storage) {
    const auto begin = reinterpret_cast<const serial_byte *>(&value);
    const auto end = begin + sizeof(value_type);
    std::copy(begin, end, storage.GetNextBytes(sizeof(value)));
  }
}
```

adapters. The three available adapters are:

`InputSerializationAdapter` This adapter is used for the serialization.

`OutputSerializationAdapter` This adapter is used for the deserialization.

`SizeAdapter` This adapter is used to determine the serialized size of the object internally.

For the serialization, the serialize buffer implements the ampersand operator (see Listing 4.11). This method has a template parameter `value_type` representing the right side of the ampersand operator. For the three-dimensional coordinate this would be a floating point type. In the nested case, this is the type of the sub-class. To enable nested serialization, the operator implementation uses the TMP feature SFINAE. In Listing 4.11, the first implementation is used for the nested serialization case. The type trait `IsSerializeable<...>` checks if a certain type `value_type` has a serialize method. If `value_type` has a serialize method, the serialization is forwarded to the serialize method of the sub-object.

In case the type has no serialize method, the second implementation is responsible. In this case, the serialization adapter is called with the value and the serialization buffer as parameter. For an input adapter this means, the value is copied from the value into the serialization buffer. For an output adapter the element will be copied out of the serialization buffer into the value variable.

Listing 4.12 shows the apply method of the `InputSerializationAdapter`. The value will be reinterpreted as bytes and copied into the serialization buffer. The

LISTING 4.13: SIZEADAPTER FOR THE COMPUTATION OF THE SERIALIZED SIZE

```
1 struct SizeAdapter {
2   template <typename value_type>
3   static void Apply(value_type &, storage_impl & storage) {
4     storage += sizeof(value_type);
5   }
6 };
```

`OutputSerializationAdapter` is similar, except that it copies data from the serialization buffer into the value.

DETERMINING THE SERIALIZED SIZE

As mentioned before, the algorithm developer only needs to define the serialization in a single method. However, for the allocation of an appropriate serialization buffer the size of the serialized object is required. The manual computation of the serialized size by the algorithm developer would be quite error-prone. The user might overestimate or underestimate the required serialized size, e.g. by adding unnecessary padding to the serialized size. The proposed interface using a single serialize method ensures that the serialized size is determined correctly using the same procedure as for the serialization itself.

This is done using the size adapter shown in Listing 4.13. Every time the input adapter attempts to copy the value into the serialization buffer, the size adapter only adds up the size of the element. Thereby the size is calculated element by element considering exactly the elements used in the serialization itself. This ensures, that the size is determined accurately and the serialized object is stored correctly.

4.2.3 ENCAPSULATION OF MESSAGE DATA

Message passing is mainly about how to transfer data and thereby about the actual message data. The interface between the algorithmic data and the message data used by MPI plays a key role for providing a high level user interface. The user interface should accept arbitrary data provided by the user and prepare it accordingly for the communication. Examples for this data are the three-dimensional coordinates or the triangular array of a multipole expansion. The data provided by the user must be handled differently for different types. Depending on the datatype it must be serialized (multipole) or a corresponding MPI datatype must be created (three-dimensional coordinate). Nevertheless, the algorithm developer should be able to send a three-dimensional coordinate or a triangular array without being bothered by the internal serialization or type registration.

MPI DATA

For the interface between MPI data and the algorithmic data it is important to repeat the requirements introduced by MPI. MPI message data consists of three parameters: a void pointer to a buffer, a count and an MPI datatype (see Figure 4.3). Additionally, MPI expects contiguously stored elements of the type reflected by

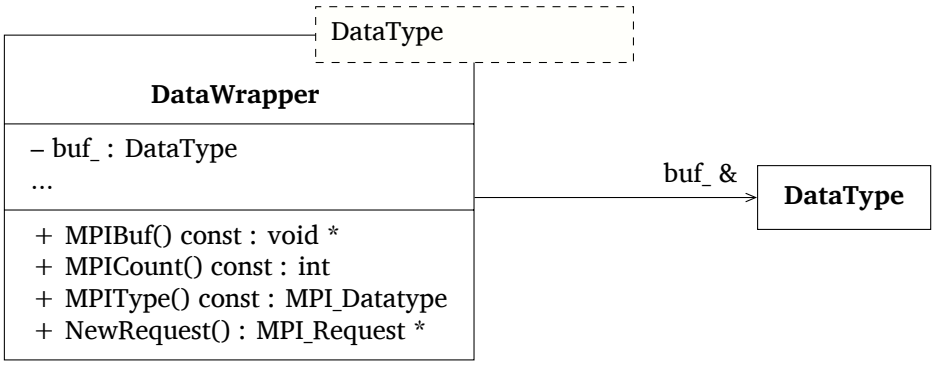


Figure 4.5: Parts of the `DataWrapper` UML class diagram. Only the internal member variables are shown.

the MPI datatype. Depending on the message size and the MPI datatype, MPI might copy the data internally into its own internal buffer. However, the data must not be deleted until a communication is finished. This makes it reasonable for the high level interface to combine the data with the request object to ensure data consistency while a communication is still pending.

Data Wrapper

The data wrapper provides the interface between algorithmic data and the underlying communication library. For the default implementation this is MPI. The data wrapper is responsible for preparing the data from the algorithm to be sent using the underlying communication library (e.g. MPI). It also includes the serialization and the datatype registration within MPI.

Let's describe the data wrapper in more detail starting with the interface to the algorithmic data. This interface is provided by various constructors of the data wrapper. For example, a data wrapper can be constructed using a standard vector. The buffer will store a reference to the first element in the vector and the counts are taken from the size of the vector. It is also possible to extend this by user-defined constructors.

During the object construction, the data wrapper checks if the provided data needs serialization. If the data wrapper finds an appropriate MPI datatype, this datatype is used otherwise the data will be serialized. If the provided data neither does have an MPI datatype nor a serialize method exists, a compile time error will be emitted.

The serialization works as follows: At first, the serialized size of a single object will be determined using the serialization library proposed. Afterwards, a serialization buffer will be allocated fitting all serialized objects. Then, the data will be iterated, object by object and serialized one after another into the serialization buffer.

The class methods for this interface can be seen in the UML class diagram in Figure 4.5. The methods work as follows:

`MPIBuf` This method returns the buffer pointer used for the communication. If

the data can be sent without serialization, it returns the pointer to the original data provided by the user. This method automatically converts the typed pointer used internally to a void pointer required by MPI. If the data needs serialization, this method returns a pointer to the internal serialization buffer. In this case, the pre-processing has serialized the data already and the serialization buffer can be used for MPI.

`MPICount` This method provides the count for the corresponding data object. If the data is sent using an MPI datatype, this method returns the number of elements in the data. In the case of serialization, this method returns the number of bytes in the serialization buffer.

`MPIType` This method provides the MPI datatype for the communication. If the data will be sent serialized, this method automatically returns `MPI_BYTE`. Otherwise this method will use the MPI type traits to resolve the MPI datatype automatically. This also involves the registration of derived datatypes if necessary.

`NewRequest` In the case of non-blocking communication, this method is used to create a new request object. The request object will be stored internally and the data wrapper can be queried for open communications. Additionally, the data wrapper waits for open request during the destruction. This avoids errors due to communication performed with deleted data.

The interface shown here, is specialized for communication with MPI. For other libraries, the methods may differ. If another library can handle typed pointers instead of void pointers, this could be adapted quickly. Additionally, other parameters could be added if it is required by the communication library.

4.2.4 Low Level MPI Wrapper

The MPI wrapper represents the interface between communication layer and the underlying MPI interface. Technically it is the only wrapper that is allowed to call MPI functions directly. All other layers above this layer must use the MPI wrapper. The MPI wrapper uses the concept of encapsulated data inside the data wrapper and automatically extracts the required MPI function parameters from it. Due to this design, the MPI wrapper is not able to handle data directly but data provided by data wrappers.

Listing 4.14 shows the implementation of the MPI wrappers version of a non-blocking receive. This example shows, how the communication done by the `RecvPointToPointWrapper` is simplified. The `RecvPointToPointWrapper` has a variadic method `RecvFromRank` (see Listing 4.16). The first parameter of this method is the source rank index. The remaining variadic parameters are forwarded to one of the various constructors of the `DataWrapper`. This data wrapper is then handed over to the MPI wrapper function. The actual non-blocking receive function requires a data wrapper, a source rank index, a tag and a communicator.

LISTING 4.14: MPIWRAPPER IRECV IMPLEMENTATION

```
template <typename recv_type, typename... AdditionalRecvTypes>
void Irecv(DataWrapper<recv_type, AdditionalRecvTypes...> & recv_data,
           int source,
           int tag,
           MPI_Comm comm) {
  MPI_Irecv(recv_data.MPIBuf(),     // void * buf
            recv_data.MPICount(),   // int count
            recv_data.MPIType(),    // MPI_Datatype datatype
            source,                 // int source
            tag,                    // int tag
            comm,                   // MPI_Comm comm
            recv_data.NewRequest() // MPI_Request * request
  );
}
```

LISTING 4.15: MPIWRAPPER ALLGATHER IMPLEMENTATION

```
template <typename send_type,
          typename... AdditionalSendTypes,
          typename recv_type,
          typename... AdditionalRecvTypes>
void Allgather(DataWrapper<const send_type, AdditionalSendTypes...> & send_data,
               DataWrapper<recv_type, AdditionalRecvTypes...> & recv_data,
               MPI_Comm comm) {
  MPI_Allgather(
      conditional_const_cast(send_data.MPIBuf()), // const void * sendbuf
      send_data.MPICount(),                       // int sendcount
      send_data.MPIType(),                        // MPI_Datatype sendtype
      recv_data.MPIBuf(),                         // void * recvbuf
      recv_data.MPICount(),                       // int recvcount
      recv_data.MPIType(),                        // MPI_Datatype recvtype
      comm                                        // MPI_Comm comm
  );
}
```

Therefore, the `RecvPointToPointWrapper` only needs to create the data wrapper and delegates everything to the MPI wrapper non-blocking receive function.

The data wrapper is responsible for the data abstraction. Hence, the call to `MPI_Irecv` takes the buffer address, the count, the MPI datatype and even the request object from the data wrapper. Whether the buffer returned from the data wrapper represents the original data or a serialization buffer of the original data is completely encapsulated in the wrapper and unknown to the MPI wrapper.

Next, let's discuss abstractions for collective functions. Listing 4.15 shows the implementation for an allgather call. In contrast to the non-blocking receive this method requires two data wrappers. One for the data to be received and one for the data to be sent. The transfer of required MPI function parameter work similar to the non-blocking receive.

4.2.5 HIGH LEVEL COMMUNICATION INTERFACE

The communication layer contains two high level communication interfaces:

- one for point-to-point communication

Listing 4.16: Example of point-to-point communication of XYZ using wrapper

```
// The actual communication
PointToPointWrapper<XYZ<double>, BlockingCommunication> pt2ptw(comm_wrapper);
pt2ptw.SendToRank(send_rank, coordinates);
pt2ptw.RecvFromRank(recv_rank, recv_coordinates);
```

Listing 4.17: Point-to-Point communication of a multipole expansion

```
// Create a Pt2Pt wrapper for Multipoles using non-blocking communication.
PointToPointWrapper<MultipoleType, NonBlockingCommunication>
    pt2ptw_multipole(comm_wrapper);
// Recv recv_omegas from recv_rank
pt2ptw_multipole.RecvFromRank(recv_rank, recv_omegas);
// Send send_omegas to all ranks in send_ranks
pt2ptw_multipole.SendToRanks(send_ranks, send_omegas);

// Wait for open requests
pt2ptw_multipole.WaitForAllRecv();
pt2ptw_multipole.WaitForAllSend();
```

- and another one for collective communication.

In the following, these two high level user interfaces will be presented in detail.

The Interface for Point-to-Point Communication

In Listing 4.4 an MPI communication example using point-to-point communication was shown. Listing 4.16 shows the equivalent MPI communication using the high level point-to-point interface of the proposed communication layer. At the beginning of the listing, a `PointToPointWrapper` for the desired type (`XYZ<double>`) is created. The second template parameter of the `PointToPointWrapper` defines whether the communication should be executed blocking or non-blocking. The available options are thereby `BlockingCommunication` and `NonBlockingCommunication`. Afterwards, the wrapper can be used for sending and receiving data of the type `XYZ<double>` using the methods `SendToRank` and `RecvFromRank`. These methods require two parameters. The send method requires a destination and the data which should be sent. Additionally, the destination for the send method can either be a single rank id or an iterable container of rank ids. This can be used for sending the same data to several destinations without providing an explicit loop. The receive method requires a source rank id and the data used for retrieving.

A more advanced use case is shown in Listing 4.17. In this example, multipole expansions from the FMM implementation are sent and received using non-blocking communication. The data-structure of the multipole expansion is more complicated then the three-dimensional coordinate example. It involves dynamic allocations and thus the creation of a reusable MPI datatype is not possible anymore. The multipole data must be serialized prior to communication. The serialization interface and the data wrapper perform this operation automatically.

Non-blocking communication always involves request objects. These objects hold open requests and are stored inside the data wrappers and can be used for

Listing 4.18: Communication of coordinates using the CollectivesWrapper

```
1 // Creating wrapper for XYZ<double> and blocking communication.
2 CollectivesWrapper<XYZ<double>, BlockingCommunication>
3     cw_particles(comm_wrapper);
4 // Setting the data for sending
5 cw_particles.BindSendData(local_particles);
6 // Setting recv data
7 cw_particles.BindRecvData(ordered_particles, local_particles.size());
8
9 // Execute an Allgather with the before set data.
10 cw_particles.Allgather(particle_dist);
```

testing or waiting. This is done with the wait methods shown at the end of the listing. However, it is not strictly required to call the wait methods explicitly. The point-to-point wrapper manages all data wrappers with open request automatically. The destructor of the point-to-point wrapper waits for all open requests ensuring a correct finalization of the communication. Nevertheless, the explicit waiting should be preferred to avoid errors like working with inconsistent data that has not been fully received yet.

The Interface for Collective Communication

The collectives wrapper provides the user interface for collective communications. Collective communication always involve data for sending or for retrieving. Some collectives require both on all participating processes (e.g. allgather, alltoall) and some require only one or the other for some processes (e.g. gather, broadcast).

Listing 4.18 shows the three-dimensional coordinates example using the high level user interface. The collectives wrapper is created for the three-dimensional coordinate type and blocking communication. In contrast to the point-to-point wrapper, the collectives wrapper requires to add the used send and receive data separately. Since not all processes may require send and receive memory, the data can be set using the methods `BindSendData` and `BindRecvData`. After setting the receive and sent data, the actual collective can be called. In this example this is done using the `Allgather` method.

4.3 Task Engine Communication Extension

The proposed communication layer provides an easy-to-use high level user interface for inter-node communication. The combination of this communication layer and the task engine imposes further challenges which need to be handled.

The task engine focuses on applications using fine-grained tasks. Classical communication usually communicates in a bulk synchronous way [113]. After a phase of computation a phase of communication follows. However, synchronizations between larger phases of an application are dissolved by the data-driven approach. This means, these synchronizations cannot be used for bulk communication anymore.

Additionally, fine-grained tasks compute the data in small portions piece by piece. Referring to the α-β model [57] the communication time t for a message of

LISTING 4.19: SENDQUEUE IMPLEMENTATION

```
class SendQueue {
  // [...]
  // Adds a new item ready for communication
  void AddReadyItem(data_type const & item) {
    std::lock_guard<std::mutex> lock(mutex_);
    ready_items_.emplace_back(item);

    // If the send buffer is full, start sending
    if (ready_items_.size() == waiting_buf_size_)
      send();
  }
  // Send the ready_items_ to the recipients_
  void send() {
    send_pt2pt_wrapper_.SendToRanks(recipients_, std::move(ready_items_));
    ready_items_.clear();
  }
  // [...]
};
```

length n, with bandwidth r and latency t_0 is given by:

$$t = t_0 + n \cdot r\,. \tag{4.1}$$

If the message is very small, the latency will dominate the time spent in the communication. This also means, communicating small chunks of data after successfully executing a single task will be slower than sending larger chunks of data after several tasks have finished. Therefore, the data will be buffered in a send queue before sending.

4.3.1 COLLECTING SEND DATA FROM DIFFERENT TASKS

The send queue is used for buffering data before communication takes place (see Listing 4.19). The send queue is configured upfront by the user by defining the chunk size and the recipients. After the computation of the data inside a task is finished, the data will be added to the send queue. When the desired chunk size is reached, the send queue will automatically send the data to the requested recipients.

Another problem arising from the modified communication scheme is the identification of the data. For bulk synchronous communication, a defined set of ordered data will be send. The corresponding receiver exactly knows the data and the order it arrives (assuming message ordering). As an example the 10 multipole expansions could have the ID starting from 100 to 109. Using the send queue, it can be ensured to send a 10 multipole expansions, but the IDs could be in an arbitrary order. Even worse, if the chunk size was set to a fixed size, it is not known which multipole expansion will be sent in which message.

A workaround could be to employ the MPI tag for communicating the identifiers. This would imply that we have to use a wildcard tag matching on the receiving end of the communication. Additionally, the tag feature itself is limited and therefore this workaround is not a sustainable solution.

The problem can be solved by extending the serialization tools discussed earlier. For the buffered send, a small meta-data object is added before the regular data in memory. This meta-data object consists of the number of elements serialized and the identifiers used internally. This is done internally, the user does not need to specify the extension with meta-data.

4.3.2 Dynamically Receiving Messages

Another issue for the task-based inter-node communication is the receiving of messages. As discussed before, multithreaded MPI introduces several performance drawbacks. Restricting the receiving to a single threads can be advantageous in this case.

A simple solution for the receiving of messages could be to post all non-blocking receives of an MPI process at the beginning and constantly check if a request is finished. However, determining all receive calls upfront might be cumbersome. Besides this, calling all receives upfront might also be a performance bottleneck. Additionally, testing a request using `MPI_Test` involves an internal critical section. Therefore it should be avoided to frequently call `MPI_Test` from concurrent threads.

The receiving mechanism offered in the task engine is restricted to a single communication thread. This thread frequently probes MPI (`MPI_Iprobe`) for new messages. If a new message is available this message will be received using a non-blocking receive. Afterwards, the communication thread will process the message further (e.g. deserialization). If the sent data is related to a data-event in the static data-flow dispatcher, the communication thread will trigger this event. Thus the static data-flow dispatcher will dispatch the event as usual (like events triggered after the computation). For the dispatcher it makes no difference if the data came from actual computation inside a task or by communication.

4.3.3 Defining Communication Schemes

For a user friendly communication interface in the task engine some things are still missing. As for the static data-flow dispatcher, the user needs to decide what to do with the data. The data-flow dispatcher allows to configure the event handlers triggered for certain events upfront. The same is required for the communication. Whenever new data is computed and ready for sending, a dispatcher needs to be configured deciding where to send the data now or later.

The static data-flow dispatcher is capable of resolving the dispatch at compile-time. Unfortunately, this is not possible for the communication dispatcher. The number of used MPI processes is not fixed at compile-time and thus is neither the distribution of work. To define a communication scheme, this however is a requirement. Therefore, the static data-flow dispatcher was extended with a communication dispatcher resulting in a hybrid (static and dynamic) data-flow dispatcher. The hybrid data-flow dispatcher functions similar to the static data-flow dispatcher. The only difference is that the communication dispatcher is called as well, after the dispatching of static events has been called. The communication scheme used for the communication dispatcher must be defined by the user upfront.

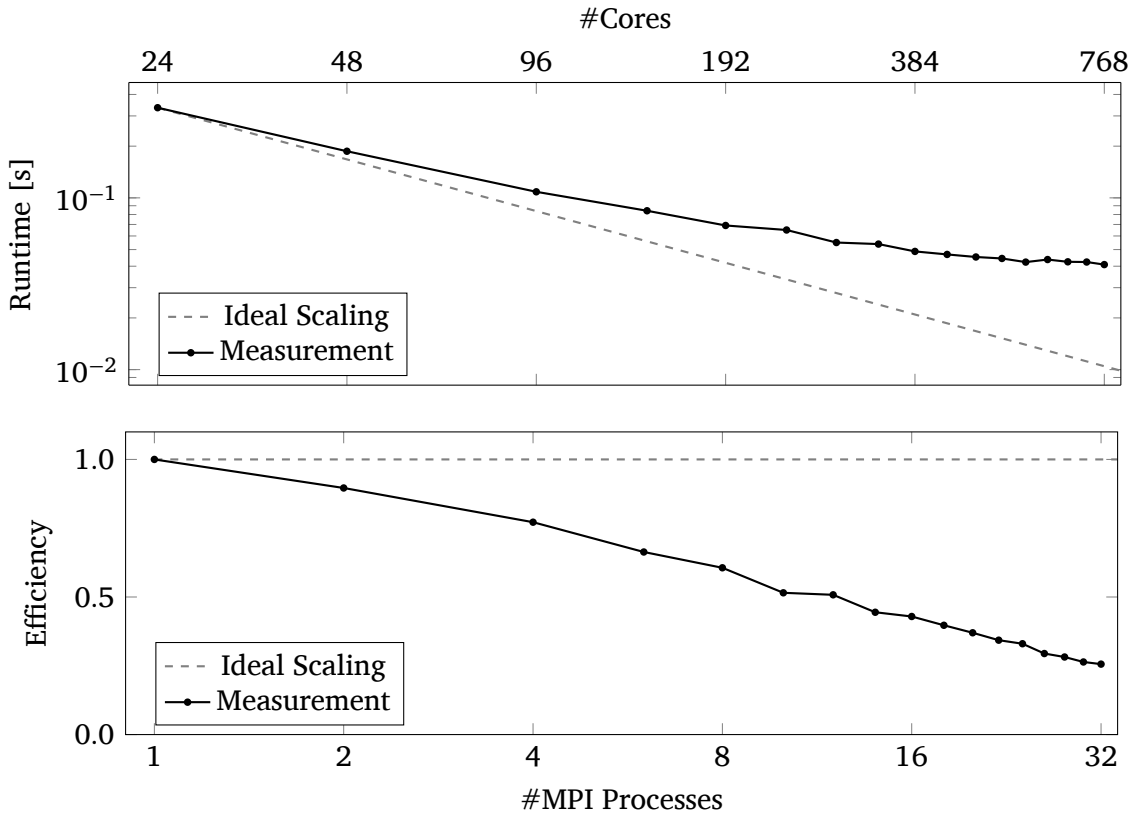


Figure 4.6: This benchmark uses the *large* particle system encompassing 103 680 particles and a multipole order of 15. It was conducted on 32 nodes of JURECA resulting in 768 cores in total.

4.4 Proof of Concept Benchmark

For the evaluation of the proposed communication extension, a proof of concept parallelization of the FMM was implemented. For the distributed computation the particles are distributed equally among the compute nodes. The compute node is responsible for computing the forces and potentials for the target particles only. For the computation a full replication scheme was used reducing the communication.

The measurements have been conducted on JURECA [73]. Each node is equipped with two Intel Xeon E5-2680 v3 processors [64] consisting of 12 cores each. Additionally, the node is equipped with 128 GiB of main memory. The nodes are connected by an InfiniBand network using a fat tree topology.

For the analysis the *large* particle system from Section 3.7 was used. It consists of 103 680 homogeneously distributed particles of silica melt [7]. The benchmark shown in Figure 4.6 was performed using up to 32 nodes resulting in a total number of 768 cores. For the particle system this results in 130 particles per core. For the measurement the computation was repeated 100 times and the 25 %-quartile was plotted (see Figure 4.6). The benchmarks exhibits a parallel efficiency of 42.9 % for 16 nodes and a parallel efficiency of 25.6 % for 32 nodes.

The minimum runtime for 16 nodes is 47 ms and for 32 nodes only 37 ms.

This approach shows a good scalability for a low number of nodes and only a moderate scalability for 32 nodes. The reason for this is the simple parallelization strategy. The full replication scheme works well for direct computation of pairwise interactions but is not sophisticated enough for the computation using the FMM. For the full replication, minor computational parts of the sequential version dominate the parallel runtime and lower the scalability (e.g. P2M and M2M). However, these limitation can be solved by implementing latency avoiding parallelization schemes [35]. Especially, these drawbacks are due to the used parallelization strategy and not due to the task engine or the communication extension.

Chapter 5

Conclusion & Outlook

This work proposed and presented a task engine for intra-node parallelization of latency-critical applications. It includes two unique components not available in any other framework, the static data-flow dispatcher and the type-driven priority scheduler. The static data-flow dispatcher can be configured at compile-time. The advantage of this approach is, that the resolving of the actual dispatch calls can be performed at compile-time, leading to the fastest dispatch possible. The type-driven priority scheduler enables the prioritization of tasks along a critical path of the parallelization. Compared to other solutions the advantage of the type-driven approach is, that the priorities are deduced automatically from the type of the task. This allows to provide an even faster insert method compared to normal bucket-based priority queues.

While the actual task engine template library implementation is extensive, its complexity can be fully hidden from the algorithm developer since only an easy-to-use interface is exposed. As stated, all compile-time features increase the performance, but more importantly the robustness and maintainability is improved. A high parallel efficiency up to 95 % on 52 shared memory cores has been achieved for a real world particle system. Even for a more challenging use case with less computation a parallel efficiency of 87.7 % on 52 shared memory cores can be reached. Such numbers are only possible because of the inherently new design of the task engine itself. In contrast to classical divide and conquer approaches in task engine like OpenMP tasks and its inherently attached limitation of fix recursion depth for the creation of new tasks, the proposed task engine does not suffer from any of these limitations. Currently executed tasks creating new tasks are not responsible for them throughout there lifetime. The responsibility is handed over to the scheduler after the creation and enqueuing of the new ready-to-execute task. Additionally, the data-driven execution model never requires to create and maintain the complete task graph upfront. Instead the much more compact data-flow graph is used to generate the static data-flow dispatcher at compile-time automatically.

For HPC applications, inter-node parallelization can not be ignored. Therefore, the task engine was extended with a communication layer. Similar to the task

engine the communication layer utilizes the separation of concerns between algorithm developers and library developers. Message passing done with MPI is not appropriate as a high level interface for algorithm developers. A minimal message passing interface only relies on the data and the desired sending or receiving counterpart. An easy-to-use high level interface was implemented in the communication layer. This communication layer is capable of handling arbitrary complex algorithmic data structure and provides a translation layer to fit the requirements of the underlying communication library like MPI. Features like type-traits and automatic serialization allow the swift translation towards the underlying communication library. To this extend, no other C++ library provides an easy-to-use high level interface. Additionally, intra-node and inter-node parallelization have been combined, providing first concepts for task-based communication using fine-grained tasks. Distributing tasks among different nodes or using implicit communication has been knowingly rejected by design. Instead tasks are only scheduled using node-local task engines and communication must be done explicitly. Currently, work-stealing over node boundaries is not permitted. The short execution times of a single time step favour a redistribution between time steps to remove potential imbalances. Preliminary measurements of the communication-enabled task engine show promising results. It was possible to scale the total runtime of a hundred thousand particle system down to 37 ms using 768 cores in total.

In general, this work shows, that the proposed strict separation of concerns works well also in HPC. The thereby introduced differentiation of two developer roles makes it easy to exchange algorithmic details without modifying the low level libraries of the task engine. On the other hand, this also allows a high flexibility on the part of the library developer in case internals like locks, allocations or scheduling strategies need to be modified. This has already been demonstrated by allowing easy to exchange MCS-locks and standard mutex locks. The stark distinction between algorithm developers and library developers relieves the algorithm developer from adapting to new hardware features frequently. This means, while the free launch from increasing CPU frequencies is still over [108], a new one is already served.

OUTLOOK

The presented communication-enabled task engine provides a strong foundation for latency-critical HPC applications. With future HPC systems in mind further enhancements are possible.

The overhead analysis revealed, that the creation of tasks requires frequent concurrent allocations. However, the default glibc allocator shows a poor performance for this use case. As an initial workaround this allocator was exchanged during runtime by pre-loading jemalloc [38]. Allocations need further exploration in case jemalloc is not allowed by the target code or other allocation strategies are required. One possibility would be the use of pool allocators like Boost.Pool [25]. While the implementation of an efficient allocator is hard, the actual exchange of the allocator in the task engine can be done with minor efforts.

Up until now, task objects in the task engine are not reused. Reusing these objects will reduce the required allocations and thereby eliminate the cause of the allocation in the first place. Again, this endeavor is not trivial, since the reuse must be implemented for different task types separately. Additionally, NUMA must be considered to avoid new overheads from falsely crossing NUMA borders using the reuse pool. In case one would go one step further towards lock-free or wait-free queues, reusing objects causes the ABA problem [55, pp. 223]. However, it is not clear, if lock-free or wait-free implementations would improve the performance significantly, since additional fine-grained synchronizations have to be introduced.

The fine-grained structure of the task engine exhibits several atomic synchronizations. E.g., for the tracking of dependencies so called dependency counters are utilized. Using a single atomic counter concurrently introduces stalls in execution due to atomic read-modify-write operations. For concurrent counting with high contention the concept of combining trees offers a better scalability [54].

For the communication layer other low level communication libraries need to be explored. It is expected to gain more performance by circumventing known MPI performance bottlenecks [29]. Additionally, other communication strategies like active messages [37] or RDMA not available in MPI should be explored.

As mentioned before, certain parts of Flynn's taxonomy have not been part of this thesis. Therefore, further work is required to combine vectorization (SIMD) with the task engine. This also benefits new architecture like GPUs since SIMD and SIMT are two sides of the same coin.

All in all, the direction taken by using modern C++ and the differentiation of developer roles shows very promising results and provides a valid baseline for other latency-critical HPC applications.

Bibliography

[1] Acar, Umut A, Blelloch, Guy E, and Blumofe, Robert D. “The data locality of work stealing”. In: *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM. 2000, pp. 1–12.

[2] Agullo, Emmanuel et al. “Task-based FMM for multicore architectures”. In: *SIAM Journal on Scientific Computing* 36.1 (2014), pp. C66–C93.

[3] Alder, Berni J and Wainwright, T E. “Studies in molecular dynamics. I. General method”. In: *The Journal of Chemical Physics* 31.2 (1959), pp. 459–466.

[4] Amarasinghe, Saman et al. “Exascale software study: Software challenges in extreme scale systems”. In: *DARPA IPTO, Air Force Research Labs, Tech. Rep* (2009).

[5] Amdahl, Gene M. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[6] Archer, Charles J and Blocksome, Michael A. *Remote direct memory access*. US Patent 8,325,633. 2012.

[7] Arnold, Axel et al. “Comparison of scalable fast methods for long-range interactions”. In: *Physical Review E* 88.6 (2013), p. 063308.

[8] Atkinson, Michael D et al. “Min-max heaps and generalized priority queues”. In: *Communications of the ACM* 29.10 (1986), pp. 996–1000. DOI: 10.1145/6617.6621.

[9] Augonnet, Cédric et al. “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures”. In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.

[10] Ayguadé, Eduard et al. “The design of OpenMP tasks”. In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (2009), pp. 404–418.

[11] Balaji, Pavan. “Interconnects”. Conference Talk. Argonne Training Program on Extreme-Scale Computing. July 2017. URL: http://press3.mcs.anl.gov/atpesc/files/2017/08/ATPESC_2017_Track-1_4_7-31_1045am_Balaji-Interconnects.pdf.

[12] Barnes, Josh and Hut, Piet. “A hierarchical O (N log N) force-calculation algorithm”. In: *nature* 324.6096 (1986), p. 446.

[13] Blechmann, Tim. *Boost. Lockfree*. 2008. URL: http://www.boost.org/doc/libs/1_63_0/doc/html/lockfree.html.

[14] Blumofe, Robert D and Leiserson, Charles E. “Scheduling multithreaded computations by work stealing”. In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[15] Boehm, Hans-J. “Threads cannot be implemented as a library”. In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 261–268.

[16] Boehm, Hans-J and Adve, Sarita V. “Foundations of the C++ concurrency memory model”. In: *ACM SIGPLAN Notices*. Vol. 43. 6. ACM. 2008, pp. 68–78.

[17] *Boost Context*. 2017. URL: `http://www.boost.org/doc/libs/1_66_0/libs/context/doc/html/index.html`.

[18] Bosilca, George et al. “DAGuE: A generic distributed DAG engine for high performance computing”. In: *Parallel Computing* 38.1-2 (2012), pp. 37–51.

[19] Bosilca, George et al. “Parsec: Exploiting heterogeneity to enhance scalability”. In: *Computing in Science & Engineering* 15.6 (2013), pp. 36–45.

[20] Bull, J Mark, Reid, Fiona, and McDonnell, Nicola. “A Microbenchmark Suite for OpenMP Tasks”. In: *International Workshop on OpenMP*. Springer. 2012, pp. 271–274.

[21] Bungartz, Hans-Joachim and Nagel, Wolfgang E. *SPPEXA: German Priority Programme Software for Exascale Computing*. 2013. URL: `http://www.sppexa.de`.

[22] Case, David A et al. “The Amber biomolecular simulation programs”. In: *Journal of Computational Chemistry* 26.16 (2005), pp. 1668–1688.

[23] Charles, James et al. “Evaluation of the Intel® Core™ i7 Turbo Boost feature”. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE. 2009, pp. 188–197.

[24] Charney, Jules G, Fjörtoft, Ragnar, and Neumann, J von. “Numerical integration of the barotropic vorticity equation”. In: *Tellus* 2.4 (1950), pp. 237–254.

[25] Cleary, Stephen. *Boost Pool Library*. 2001.

[26] *Condition Variable*. URL: `http://en.cppreference.com/w/cpp/thread/condition_variable`.

[27] Dachsel, Holger, Hofmann, Michael, and Rünger, Gudula. “Library support for parallel sorting in scientific computations”. In: *European Conference on Parallel Processing*. Springer. 2007, pp. 695–704.

[28] Dagum, Leonardo and Menon, Ramesh. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[29] Dang, Hoang-Vu, Snir, Marc, and Gropp, William. “Eliminating contention bottlenecks in multithreaded MPI”. In: *Parallel Computing* 69 (2017), pp. 1–23.

[30] Dang, Hoang-Vu, Snir, Marc, and Gropp, William. “Towards millions of communicating threads”. In: *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM. 2016, pp. 1–14. DOI: 10.1145/2966884.2966914.

[31] Darden, Tom, York, Darrin, and Pedersen, Lee. “Particle mesh Ewald: An N log (N) method for Ewald sums in large systems”. In: *The Journal of chemical physics* 98.12 (1993), pp. 10089–10092.

[32] Desrochers, Cameron. *moodycamel::ConcurrentQueue*. 2015. URL: https://github.com/cameron314/concurrentqueue.

[33] Dial, Robert B. “Algorithm 360: Shortest-path forest with topological ordering [H]”. In: *Communications of the ACM* 12.11 (1969), pp. 632–633. DOI: 10.1145/363269.363610.

[34] Driesen, Karel and Hölzle, Urs. “The direct cost of virtual function calls in C++”. In: *ACM Sigplan Notices*. Vol. 31. 10. ACM. 1996, pp. 306–323.

[35] Driscoll, Michael et al. “A communication-optimal n-body algorithm for direct interactions”. In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE. 2013, pp. 1075–1084.

[36] Dursi, Jonathan. *HPC is dying, and MPI is killing it*. 2015. URL: https://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it.html.

[37] Eicken, TV et al. “Active messages: a mechanism for integrated communication and computation”. In: *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*. IEEE. 1992, pp. 256–266.

[38] Evans, Jason. “A scalable concurrent malloc implementation for FreeBSD”. In: *Proc. of the bsdcan conference, ottawa, canada*. 2006.

[39] Faxén, Karl-Filip. “Wool-a work stealing library”. In: *ACM SIGARCH Computer Architecture News* 36.5 (2008), pp. 93–100.

[40] Flynn, Michael J. “Some computer organizations and their effectiveness”. In: *IEEE transactions on computers* 100.9 (1972), pp. 948–960.

[41] *Fmsolvr*. 2018. URL: http://fmsolvr.org.

[42] Forum, Message Passing Interface. *MPI: A Message Passing Interface Version 1.0*. May 1994. URL: http://mpi-forum.org/docs/.

[43] Forum, Message Passing Interface. *MPI: A Message Passing Interface Version 3.1*. June 2015. URL: http://mpi-forum.org/docs/.

[44] Franke, Hubertus, Russell, Rusty, and Kirkwood, Matthew. “Fuss, futexes and furwocks: Fast userlevel locking in linux”. In: *AUUG Conference Proceedings*. Vol. 85. AUUG, Inc. 2002.

[45] Frazer, Kelly A. “Decoding the human genome”. In: *Genome research* 22.9 (2012), pp. 1599–1601.

[46] Fukushige, Toshiyuki et al. “A highly parallelized special-purpose computer for many-body simulations with an arbitrary central force: MD-GRAPE”. In: *The Astrophysical Journal* 468 (1996), p. 51.

[47] Fürlinger, Karl, Fuchs, Tobias, and Kowalewski, Roger. “DASH: a C++ PGAS library for distributed data structures and parallel algorithms”. In: *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*. IEEE. 2016, pp. 983–990.

[48] El-Ghazawi, Tarek and Smith, Lauren. “UPC: unified parallel C”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 27.

[49] Goldthwaite, Lois. “Technical report on C++ performance”. In: *ISO/IEC PDTR* 18015 (2006).

[50] Greengard, Leslie and Rokhlin, Vladimir. “A fast algorithm for particle simulations”. In: *Journal of Computational Physics* 73.2 (1987), pp. 325–348. DOI: 10.1016/0021-9991(87)90140-9.

[51] Gregor, Douglas and Troyer, Matthias. *Boost.mpi*. 2006.

[52] Gropp, William and Thakur, Rajeev. “Issues in developing a thread-safe MPI implementation”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2006), pp. 12–21.

[53] Grubmüller, Helmut, Dachsel, Holger, and Hess, Berk. *GROMEX: Unified long-range electrostatics and dynamic protonation for realistic biomolecular simulations on the Exascale*. 2013. URL: http://www.mpibpc.mpg.de/grubmueller/sppexa.

[54] Herlihy, Maurice, Lim, Beng-Hong, and Shavit, Nir. “Scalable concurrent counting”. In: *ACM Transactions on Computer Systems (TOCS)* 13.4 (1995), pp. 343–364.

[55] Herlihy, Maurice and Shavit, Nir. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[56] Hess, Berk et al. “GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation”. In: *Journal of chemical theory and computation* 4.3 (2008), pp. 435–447.

[57] Hockney, Roger W. “The communication challenge for MPP: Intel Paragon and Meiko CS-2”. In: *Parallel computing* 20.3 (1994), pp. 389–398.

[58] Hoefler, Torsten and Belli, Roberto. “Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2015, p. 73.

[59] Hunt, Andrew and Thomas, David. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[60] Hyafil, L. and Rivest, R.L. *Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems*. Laboratoire de Recherche: Rapport de recherche. IRIA, 1973.

[61] *Intel TBB Data Flow and Dependence Graphs*. 2017. URL: `https://software.intel.com/en-us/node/517340`.

[62] *Intel Xeon Phi x200 Product Family*. 2017. URL: `https://ark.intel.com/products/series/92650/Intel-Xeon-Phi-x200-Product-Family`.

[63] *Intel Xeon Platinum 8170 Processor*. 2017. URL: `https://ark.intel.com/products/120506/Intel-Xeon-Platinum-8170-Processor-35_75M-Cache-2_10-GHz`.

[64] *Intel Xeon Prozessor E5-2680 v3*. 2014. URL: `https://ark.intel.com/de/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz`.

[65] *Intel Xeon Scalable Processors*. 2017. URL: `https://ark.intel.com/products/series/125191/Intel-Xeon-Scalable-Processors`.

[66] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.)

[67] Johnson, Eric E. “Completing an MIMD multiprocessor taxonomy”. In: *ACM SIGARCH Computer Architecture News* 16.3 (1988), pp. 44–47.

[68] Josuttis, Nicolai M. *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012.

[69] Kabadshow, Ivo. *Periodic boundary conditions and the error-controlled fast multipole method*. Vol. 11. Forschungszentrum Jülich, 2012.

[70] Kaiser, Hartmut et al. “HPX: A Task Based Programming Model in a Global Address Space”. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM. 2014, p. 6. DOI: `10.1145/2676870.2676883`.

[71] Kale, Laxmikant V and Krishnan, Sanjeev. “CHARM++: a portable concurrent object oriented system based on C++”. In: *ACM Sigplan Notices*. Vol. 28. 10. ACM. 1993, pp. 91–108. DOI: `10.1145/167962.165874`.

[72] Klenk, Benjamin and Fröning, Holger. “An Overview of MPI Characteristics of Exascale Proxy Applications”. In: *International Supercomputing Conference*. Springer. 2017, pp. 217–236. DOI: 10.1007/978-3-319-58667-0_12.

[73] Krause, Dorian and Thörnig, Philipp. “JURECA: General-purpose supercomputer at Jülich Supercomputing Centre”. In: *Journal of large-scale research facilities JLSRF* 2 (2016), p. 62.

[74] Lamport, Leslie. “A new solution of Dijkstra’s concurrent programming problem”. In: *Communications of the ACM* 17.8 (1974), pp. 453–455. DOI: 10.1145/361082.361093.

[75] Langr, Jeff. *Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better*. Pragmatic Bookshelf, 2013.

[76] Liskov, Barbara H and Wing, Jeannette M. “A behavioral notion of subtyping”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994), pp. 1811–1841. DOI: 10.1145/197320.197383.

[77] Martin, Robert C. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[78] Mellor-Crummey, John M and Scott, Michael L. “Algorithms for scalable synchronization on shared-memory multiprocessors”. In: *ACM Transactions on Computer Systems (TOCS)* 9.1 (1991), pp. 21–65.

[79] Messina, Paul. “The Exascale Computing Project”. In: *Computing in Science & Engineering* 19.3 (2017), pp. 63–67.

[80] Meyer, Bertrand. *Object-oriented software construction*. Vol. 2. Prentice hall New York, 1988.

[81] Meyers, Scott. “The most important design guideline?” In: *IEEE Software* 21.4 (2004), pp. 14–16. DOI: 10.1109/MS.2004.29.

[82] Milewski, Bartosz. *Supercomputing: An Industry in Need of a Revolution*. 2011. URL: https://bartoszmilewski.com/2011/11/21/supercomputing-an-industry-in-need-of-a-revolution/.

[83] Molka, Daniel, Hackenberg, Daniel, and Schöne, Robert. “Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer”. In: *Proceedings of the workshop on Memory Systems Performance and Correctness*. ACM. 2014, p. 4.

[84] Moore, Gordon E. *Cramming More Components Onto Integrated Circuits, Electronics,(38) 8*. 1965.

[85] Morgenstern, Laura. *A NUMA-Aware Task-Based Load-Balancing Scheme for the Fast Multipole Method*. 2017. DOI: 10.13140/RG.2.2.35575.93603.

[86] *MPICH*. URL: http://www.mpich.org.

[87] Munro, J Ian and Suwanda, Hendra. “Implicit data structures for fast search and update”. In: *Journal of Computer and System Sciences* 21.2 (1980), pp. 236–250. DOI: 10.1016/0022-0000(80)90037-9.

[88] Nelson, Mark T et al. “NAMD: a parallel, object-oriented molecular dynamics program”. In: *The International Journal of Supercomputer Applications and High Performance Computing* 10.4 (1996), pp. 251–268.

[89] Nichols, Bradford, Buttlar, Dick, and Farrell, Jacqueline. *Pthreads programming: A POSIX standard for better multiprocessing.* ” O’Reilly Media, Inc.”, 1996.

[90] Numrich, Robert W and Reid, John. “Co-Array Fortran for parallel programming”. In: *ACM Sigplan Fortran Forum*. Vol. 17. 2. ACM. 1998, pp. 1–31.

[91] Olivier, Stephen L et al. “OpenMP task scheduling strategies for multicore NUMA systems”. In: *The International Journal of High Performance Computing Applications* 26.2 (2012), pp. 110–124.

[92] *OpenMPI*. URL: https://www.open-mpi.org.

[93] *Overload resolution*. 2017. URL: http://en.cppreference.com/w/cpp/language/overload_resolution.

[94] Papamarcos, Mark S and Patel, Janak H. “A low-overhead coherence solution for multiprocessors with private cache memories”. In: *ACM SIGARCH Computer Architecture News* 12.3 (1984), pp. 348–354. DOI: 10.1145/773453.808204.

[95] Pellegrini, Simone, Prodan, Radu, and Fahringer, Thomas. “A Lightweight C++ Interface to MPI”. In: *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference On*. Vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 3–10. DOI: 10.1109/PDP.2012.42.

[96] Pheatt, Chuck. “Intel® threading building blocks”. In: *Journal of Computing Sciences in Colleges* 23.4 (2008), pp. 298–298.

[97] Podobas, Artur, Brorsson, Mats, and Faxén, Karl-Filip. “A comparison of some recent task-based parallel programming models”. In: *3rd Workshop on Programmability Issues for Multi-Core Computers*. 2010.

[98] Raffenetti, Ken. “Next Generation MPICH: What to Expect – Lightweight communication and much more!” Conference Talk. Intel HPC Developer Conference. Nov. 2017. URL: https://www.intel.com/content/dam/www/public/us/en/documents/presentation/next-generation-mpich.pdf.

[99] Ramey, Robert. *Boost Serialization*. 2004. URL: http://www.boost.org/doc/libs/1_66_0/libs/serialization/doc/index.html.

[100] Reed, Daniel A et al. *Computational science: Ensuring America's competitiveness*. Tech. rep. PRESIDENT'S INFORMATION TECHNOLOGY ADVISORY COMMITTEE ARLINGTON VA, 2005.

[101] Rhee, Young Min et al. "Simulations of the role of water in the protein-folding mechanism". In: *Proceedings of the National Academy of Sciences of the United States of America* 101.17 (2004), pp. 6456–6461.

[102] Seo, Sangmin et al. "Argobots: A Lightweight Low-Level Threading and Tasking Framework". In: *IEEE Transactions on Parallel and Distributed Systems* (2017). DOI: 10.1109/TPDS.2017.2766062.

[103] Shaw, David E et al. "Anton, a special-purpose machine for molecular dynamics simulation". In: *Communications of the ACM* 51.7 (2008), pp. 91–97.

[104] Snir, Marc. "MPI is too High-Level; MPI is too Low-Level". Conference Talk. MPI Symposium. Sept. 2017. URL: http://www.mcs.anl.gov/mpi-symposium/slides/marc_snir_25yrsmpi.pdf.

[105] Squyres, Jeffrey M, Saphir, Bill, and Lumsdaine, Andrew. "The design and evolution of the MPI-2 C++ interface". In: *International Conference on Computing in Object-Oriented Parallel Environments*. Springer. 1997, pp. 57–64. DOI: 10.1007/3-540-63827-X_44.

[106] Stroustrup, Bjarne. "Foundations of C++". In: *European Symposium on Programming*. Springer. 2012, pp. 1–25.

[107] Stroustrup, Bjarne. "Software development for infrastructure". In: *IEEE Computer* 45.1 (2012), pp. 47–58.

[108] Sutter, Herb. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[109] Tallent, Nathan R and Mellor-Crummey, John M. "Effective performance measurement and analysis of multithreaded applications". In: *ACM Sigplan Notices*. Vol. 44. 4. ACM. 2009, pp. 229–240. DOI: 10.1145/1594835.1504210.

[110] Thakur, Rajeev and Gropp, William. "Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2007, pp. 46–55.

[111] *TOP500*. 2018. URL: https://www.top500.org.

[112] Trottenberg, Ulrich, Oosterlee, Cornelius W, and Schuller, Anton. *Multigrid*. Academic press, 2000.

[113] Valiant, Leslie G. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[114] Vandevoorde, D., Josuttis, N.M., and Gregor, D. *C++ Templates: The Complete Guide*. Pearson Education, 2017. ISBN: 9780134778747. URL: http://www.tmplbook.com/.

[115] Veldhuizen, Todd L. “C++ Templates are Turing Complete”. In: (2003).

[116] Wall, Mike. *Touchdown! Huge NASA Rover Lands on Mars*. 2012. URL: https://www.space.com/16932-mars-rover-curiosity-landing-success.html.

[117] White, Christopher A and Head-Gordon, Martin. “Rotating around the quartic angular momentum barrier in fast multipole method calculations”. In: *The Journal of Chemical Physics* 105.12 (1996), pp. 5061–5067.

[118] Williams, Anthony. *C++ concurrency in action: practical multithreading*. Manning Publ., 2012.

[119] Witman, Sarah. *Meet the Computer Scientist You Should Thank For Your Smartphone’s Weather App*. 2017. URL: https://www.smithsonianmag.com/science-nature/meet-computer-scientist-you-should-thank-your-phone-weather-app-180963716/.

[120] *World Record: Quantum Computer with 46 Qubits simulated*. 2017. URL: http://www.fz-juelich.de/SharedDocs/Pressemitteilungen/UK/EN/2017/2017-12-15-world-record-juelich-researchers-simulate-quantum-computer.html.

[121] Yamamoto, Keiji et al. “The K computer operations: experiences and statistics”. In: *Procedia Computer Science* 29 (2014), pp. 576–585.

[122] Ying, Lexing et al. “A new parallel kernel-independent fast multipole method”. In: *Supercomputing, 2003 ACM/IEEE Conference*. IEEE. 2003, pp. 14–14.

[123] Yokota, Rio and Barba, Lorena A. “A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems”. In: *The International Journal of High Performance Computing Applications* 26.4 (2012), pp. 337–346.

[124] Zink, Mareike and Grubmüller, Helmut. “Primary changes of the mechanical properties of Southern Bean Mosaic Virus upon calcium removal”. In: *Biophysical journal* 98.4 (2010), pp. 687–695.